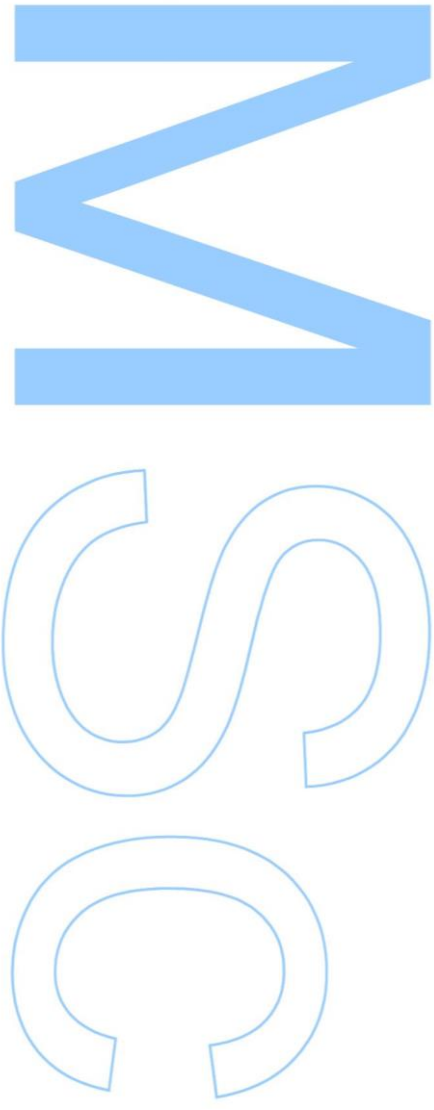
Backend for a Ticketing System

Rafael Borges de Almeida
Mestrado Integrado em Engenharia de Redes e Sistemas Informáticos
Departamento de Ciência de Computadores
2018

Orientador
Rui Pedro de Magalhães Claro Prior, Professor Auxiliar,
Faculdade de Ciências da Universidade do Porto



U.PORTO

FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Todas as correções determinadas pelo júri, e só essas, foram efetuadas.

O Presidente do Júri,

Porto, _____/_____/_____

Abstract

With society slowly but steadily walking towards an increasingly digitized world, some services have been lagging behind on their response to this societal transition. Nowadays, there is an observable increase on digital alternatives for multiple facets of daily life, such as transportation, news, and shopping. An area which has been noticeably lacking in digital solutions is the access to event venues, where paper tickets are still the norm. This work focuses on the design and implementation of an e-ticketing system, in an effort to fill in this the gap between the current solutions for box offices and other commercial services. The design will be focused on satisfying the basic requirements and needs of said systems whether they be functionalities from the event organizer point of view or from the end-user. Increasingly, as progress was made, further quality of life functions were explored and, completely or partially, implemented, in an attempt for the prototype created to be as deployment ready as possible. In this note, there was some integration performed with an existing mobile technology, more specifically, the usage of a mobile payment system in order to offer a native money transaction support. Furthermore, the prototype will be developed from the start with security being a concern, from introduction of well known good practices, for example, use of certificates for e-ticket creation and communications to including some more innovate features such as implementing a multi factor authentication for e-ticket validation. At the conclusion, we plan to have a working prototype, that should satisfy the basic needs of any event agency using it, such as events creation, ticket purchase and validation of those tickets.

Resumo

Com a sociedade andando lenta mas firmemente em direção a um mundo digitalizado, alguns serviços têm ficado para trás na sua resposta para esta transição social. Hoje em dia, há um aumento observável de alternativas digitais para múltiplas facetas da vida cotidiana, como transporte, notícias e compras. Uma área visivelmente tem sofrido falta de soluções digitais é o acesso a locais de eventos, onde o ingressos a papel ainda são a norma. Este trabalho concentra-se na concepção e implementação de sistema de bilhética eletrônico, em um esforço para preencher esta lacuna entre as soluções atuais para bilheterias e outros serviços comerciais. O design será focado em satisfazer os requisitos básicos e necessidades dos referidos sistemas sejam funcionalidades do ponto de vista do organizador do evento ou do usuário final. Progressivamente, conforme o progresso foi feito, outras funcionlidades opcionais são exploradas e, integralmente ou parcialmente, implmentadas na tentativa de que o protótipo criado esteja o mais proximo de produção que for possivel. Nesta nota, houve alguma integração realizada com uma tecnologia móvel existente, mais especificamente, o uso de um sistema de pagamento móvel, a fim de oferecer uma solução nativa a transações e pagamentos. Além disso, o protótipo será desenvolvido a partir do começo com a segurança sendo uma preocupação, desde a introdução de boas práticas conhecidas, como o uso de certificados para criação dos bilhetes digitais e comunicações, e incluir alguns recursos mais inovadores, como por exemplo a implementação de uma autenticação multi-fator para validação dos bilhetes. Na conclusão, planeamos ter um protótipo funcional que satisfaça as necessidades básicas de qualquer agência de eventos, como criação de eventos, compra de ingressos e validação desses ingressos.

Agradecimentos

Gostaria de agradeceder a orientação e a ajuda do Professor Rui Prior ao longo da dissertação. Gostava igualmente de agradecer ao Instituto de Telecomunicações por acolher este trabalho.

Aos meus amigos por toda a ajuda durante a escrita desta dissertação e aos meus pais pelo apoio durante todo o meu percurso académico

À Joana por todo o apoio e ajuda neste ultimo ano.

Contents

List of Tables

List of Figures

Listings

Acronyms

API Application Programming Interface

CSR Certificate Signing Request

HMAC Hash-based Message Authentication Code

HTTP HyperText Transfer Protocol

HTTPS Hyper Text Transfer Protocol Secure

ID Identification

IPC Inter-Process Communication

JSON JavaScript Object Notation

NFC Near-Field Communication

POS Point of Sale

QR Quick Response

REST Representational State Transfer

SOAP Simple Object Access Protocol

SQL Structured Query Language

SSL Secure Sockets Layer

TOTP Time-based One-Time Password

Chapter 1

Introduction

We, as consumers, tend to choose options that seem simpler and safer, that offer a higher level of security and privacy [23], with a tendency to prefer solutions that can be achieved ”on the go”, without the use of a considerable amount of time to complete. E-commerce has explored this by offering its consumers a wide variety of choices directly to their homes, and offering convenience for several services. One of the most common services is the purchase of airline tickets[15]. Following E-commerce and due to the widespread availability of increasingly powerful mobile devices, M-commerce offers users the capability of buying and requesting services while on the move[9]. Many of these services can be translated into an e-ticket system, independently of the context of this service. Several e-ticket systems are already in use, such as the current use of Near-Field Communication (NFC) cards on public transportation[16]. However, the development of these systems is usually restricted to the context of the services implementing the e-tickets. Thus, if a e-ticket system that is completely independent of context could be created, it would present service providers with an easy access point for the association of their services to a ticket. Such a system would need to meet the reliability, security and privacy features that consumers want. It would also open a market to those service providers to which said market has been out of reach.

1.1 Context

1.1.1 e-ticket

An e-ticket, or electronic ticket, represents a contract between the owner of the ticket and the service provider. The user with ownership of the ticket, by presenting it to the respective validation system and subsequent corroboration from this system, is entitled to the service that is represented by said ticket. That service would not only vary depending on the context, but also the ticket itself which can have single use or multiple uses depending upon the validation conditions. Security is crucial as it is needed to establish ownership and to prevent forgery.

1.1.2 e-ticket System

According to Payeras-Capellà[22], there is a general consensus about the main actors of such systems:

- User: The owner of the e-ticket issued by the system. When necessary, he is capable of sending the ticket for validation in order to gain access to the respective service.
- Issuer: Usually the issuer would be an intermediary or the service provider itself, but in this work, the issuer is the system being developed and acts as a trusted intermediary between the user and service provider. It should be capable of issuing any ticket categories as registered by the service provider and requested by the user.
- Service Provider: Should be capable of registering the services with the system and also able to validate any ticket the user presents.

On the other hand, the phases which make up an e-ticket system are not well defined. The most common definition is a 3 phase system: e-ticket payment, issue and validation. However, also according to Payeras-Capellà[22], some authors group the payment and issue phases into a single one while others propose the existence of a registration phase in order to identify and authenticate the users. This disagreement is mostly due to the different services that are implementing the e-ticket system.

1.2 Motivation

Security and privacy are two of the main concerns on any M-commerce application[14]. If we give service providers a choice of a system that would work out-of-the-box while satisfying the security and privacy needs, it would provide them a reliable alternative to current traditional paper based systems. Maintaining a bridge from new systems to old ones is also a concern, though, in order to promote an easier transition. Usually, these systems are already an alternative provided by bigger ticketing organizations. However in order to be able to use them, service providers are charged significant fees which are a hard option for smaller providers. Fortunately, the evolution of computational power, small but powerful computing devices, like the emergence of single-board computers and strong mobile devices, allow for the development of a system capable of performing the needs of an e-ticketing system without the need for a heavy investment as well as minimal involvement of third-parties. This would facilitate the access of e-ticket systems to smaller providers.

As a consequence to this facilitated access, users would have an increased contact with newer technologies being used by a wider range of services. This should increase their confidence and openness to these technologies. This could, in turn, further help the transition from traditional ticketing systems to the new system.

1.3 Objectives

In order to steer this work, we propose a few general goals, on top of later defined system requirements. The two main goals to be achieved by the end of this work are: first, that the system should be a deployment ready prototype which means that, aside from possible configurations, its deployment should be done without any need of further modifications and the second goal, involves keeping the system hardware requirements in consideration so that, at deployment, only minimal hardware requirements have to be met.

Other goals are to satisfy the notion of an e-ticket system as described above. For this purpose, again as described, three actors to the system will be considered: the user, the issuer and the service provider as well as small core functionality goals associated with each actor.

- User
  - Access to tickets descriptions;
  - Ability to acquire tickets;
  - Access to bought tickets.
- Issuer
  - Issue tickets to the user;
  - Perform payment requests.
- Service Provider
  - Registration of ticket categories;
  - Validation of tickets.

More functionality will be added in order to satisfy system requirements, but those should not be required for the core functionally.

1.4 Summary

In chapter 2 we will look at a few published articles on the subject and similar practical applications. Following onto chapter 3 in which we will analyze all possible ways that such system can be used, independently of any implemented infrastructure. Chapter 4 will describe all functional and non-functional requirements on the system, which are used to satisfy some of the cases described on chapter 3. These requirements are used to determine the system architecture as described on chapter 5, in which we will start making decisions based on how the different components will work and communicate. Chapter 6 describes the different implementation choices, including database designs, development environments and system configurations. In

the last two chapters, 7 and 8, the results of any tests performed on the system will be shown, followed by, a comparison between the developed prototype and the planned objectives, as well as, an inspection of any problems that were encountered during the tests or development of this work.

Chapter 2

State of the Art

2.1 Background

There are some proposed, and a few already implemented, solutions to e-ticketing systems. One of them, which is proposed by Anita Chaudhari[11], implements a simple system for train transportation where the users register an account to which they can add credit to, for the purpose of buying tickets for the destination using the application which in turn uses the Near-Field Communication (NFC) communication point to determine the source of the communication. The user can then preview the ticket and save an image with the Quick Response (QR) code for later validation. This simple implementation incurs the inconvenience of requiring the user to already be at the point of origin in order to buy the ticket. Also, being able to save an image of the QR code for later use, allows for the possibility that the ticket be handed to a third party without the service provider's knowledge, creating multiple security problems.

Another proposal is made by Cosmina Ivan[17], also in the context of public transportation, which presents a system in a later stage of development and with some usability testing. The proposal is focused on increasing the mobility of the user by allowing said user to buy tickets exclusively through the mobile application, unlike in the aforementioned work, and the use of NFC tags to validate the tickets. During the validation process, the user taps a NFC-enabled device with the ticketing application on it against the NFC tag. This tag transfers to the device information relative to the transportation and in turn, the device communicates this information to the application server. This validation process does not prevent any foul play on the user side, for example, the use of a stub application that reads the information from the NFC tag but does not communicate with the application server.

A further paper describes an e-ticket system with small events in mind[12]. It focuses on a mobile application where the user can search for and buy tickets for events, and on utilizing an offline validation system. To validate, the user prior to the event, downloads the ticket information signed with a event specific private key. At the venue, the user uses NFC communication to transfer that ticket to a validator which checks the integrity of the ticket using the public key

and resorting to a tracking system in order to avoid the reuse of tickets. This system has the advantage that the validator can be any NFC capable device, allowing for the use of mobile phones and reducing the costs of operation.

2.2 Related Work

Similar systems to the one being studied are already being deployed in several services. For instance, a recent release at the time of writing, is the application *Anda*[1] developed by *Faculdade de Engenharia da Universidade do Porto* and *Transportes Intermodais do Porto*. This application allows the user to choose which type of single use ticket to buy, pay using direct debit or credit card with later validation of the ticket using NFC, using the already existing framework. The main limitation of the application, is the fact that it requires the cellphone to have NFC hardware, which limits the amount of users.

Other services using similar approaches, are the airline companies. Taking as an example the Ryanair application[3] which is capable of listing available flights, allows the user to purchase tickets, and also display them to be verified during check-in using a QR. The Portuguese airline *Transportes Aéreos Portugueses*[4], also has an application for this purpose, although searching for a flight using said application, opens the smartphone's browser taking the user to a website instead of performing the operation in application, which suggests that there is no means to use the ticket through the application.

Another recent example is the *placard* betting platform solution[8]. It is a mobile application which allows users to electronically create betting tickets that are translated into a QR code. This code is then used by an authorized mediator to register the bet into the system. However, this application does not keep any history of bets, and does not allow direct payment or perform payouts.

A system that should be more contextually similar to this work is the application used by the company *Bilheteira Online*[2] which is an application for a ticket office. This application would have very similar if not the same requirements that will be presented later on. However, this application usage seems to be limited to only inform authorized agents with an access token, information about an event such as entries and exits of the event enclosure. Instead, the current digital solution is a online website, where we can search for events and acquire tickets, which are then emailed to the user. The ticket should then be printed and shown at the entrance, where the validators read a barcode.

Most of these implementations consist on very similar processes which can be, in a general way, translated into converting ticket information into a message, which is then, normally, transported by use of NFC, QR or a printed ticket. In case of the QR code, the code itself can be used when printed out, shown from an received email or simply stored in an image for later use. This indicates that the code only retains information to identify the ticket. There is no attempt automatically performed by the underlying systems, to determine if the ticket itself

is a copy or, in case shown using a smartphone, the legitimacy of the user claiming to be the owner. These verifications, are performed on the side by the use of an identification card and checking if the ticket in question was already used, which would leave open the question of which ticket was forged. The use of printed ticket also supports this notion, since it is not using any digital device, any attempt automatic multi layer validation is not possible, without further input. Unfortunately, due to these implementations not being open-sourced, we cannot draw any more conclusions of how these systems work in the background or further analyze the NFC communications.

Contrasting to company specific solutions, there is an open-source protocol[5] which studies the existing problem of ticket resellers buying tickets from primary locations and reselling them at a very high markup over market price. It then proposes a protocol based on Ethereum's smart contracts in order to establish a chain of ownership, allowing end-users to verify the tickets' legitimacy and allowing service providers, which can be considered the primary ticket sellers, to control ticket resale prices and even charging fees. The main contrasting point between this alternative and this document's proposition, is that the protocol only covers an alternative to ticket signing, leaving open a third-party layer in which developers create the actual e-ticket application.

Chapter 3

Use Cases

Context, usage requirements and functionalities are needed in order for the planning of any software development while maintaining it updatable for future demands. Therefore, before exploring any further, we should describe, and analyze all possible uses within the context, even if at first not relevant or not within the short-term scope of development.

These limitations will directly affect in what ways we can perform the vending, storage and validation of our e-tickets and in this chapter, we will describe and analyze our options, while also looking at end-to-end usage.

3.1 Ticket storage

Storing the ticket directly impacts upon the kind of security we can offer to the user's side of the system. Quick example of this limitation is the existence or lack thereof of computing power which would allow for more complex and possibly safer alternatives, more precisely if the storage is based on cellphone versus a piece of paper.

The storage environments that will be analyzed are:

- paper
- smart card
- smartphone

And the pros and cons will be based on these capabilities:

- computability
- reusability
- security

- user comfort
- points of failure

3.1.1 Paper

Although seemingly counter-productive and against the idea of e-tickets, paper tickets sold at physical box offices or printed at home (via some website), can be connected to our back-end and work natively with all other functionalities, by simply having a printed out Quick Response (QR) code which is associated with a key that would give access and reference to any other required information. This method is appropriate for static, single use tickets despite the zero computation capabilities on the client side. It severely limits security, allowing failures such as the theft and copy of the ticket. It would also be a limiting factor for validators as it would require that they have a camera or some other means to visually read the information printed on the paper. In spite of these issues, it can arguably be considered the most comfortable method to use due to its similarity with traditional tickets and the fact it does not need any extra hardware.

3.1.2 Smart Cards

Similar to today bus passes, smart cards can be charged with keys corresponding to tickets, allowing for the re-usability of the same card by later recharges. Being on these cards, also permits the use of some computability should its use be needed on the security protocols while validating the ticket, although it would imply that validators have access to means of reading these cards. Some user comfort is lost if, in the context of use, there would be any reason to select the ticket to be used. Other than that, it can be argued that it is one of the most comfortable methods to use due to the public's adaptation using similar systems on public transportation.

3.1.3 Smartphone

The on demand online access, high computational power and interactivity of today phones makes them the perfect storage for e-tickets. Online access allows tickets purchase and any possibly required management, along with other security options that would require bi-directional communication with the server. This might be the least comfortable option for users since it requires the use and familiarity with more recent technology and this method being the most contrasting one from a typical ticket but might also be the safest since a cellphone is less likely to be lost or stolen than a piece of paper or a card. Furthermore, if the internet is used, it adds a whole new security concern during communication with the server.

3.2 Buying

Tickets transactions can be summarized onto two possibilities, via a physical or digital box office. Usually, we associate digital box offices with e-tickets, however both methods share some common ground and are not necessarily limiting on the choice of the ticket's storage. A physical box office, can sell an e-ticket simply by receiving the customer's information (e-mail for example) and associating this with the ticket key. And the reverse can also apply where the customer buys a ticket through a website and prints the ticket at home into a QR code. The main limitation affecting storage, would be the use of smart cards, since if the ticket is bought at home and the customer wishes to charge a smart card, it would require an Near-Field Communication (NFC) capable device.

Furthermore, the means of selling the ticket should not introduce compromises to any security features. The security protocols applied to the selling of the ticket at a website or through an app, would need to be similarly applied to physical box offices.

3.3 Validation

Reliability and speed are crucial during ticket validation. Ideally there would be a permanent internet connection with the infrastructure, turning the validator into a instrument that would only read the ticket (paper, card or cellphone) and transmit to the operator the result of the read. In a real world situation, internet connection may fail or suffer disruptions, therefore some form of offline validation is required so that, even if it wouldn't be as secure, there would be no immediate disruption to the service.

Validation, in general, should be able to ensure two things: ticket legitimacy and ownership. Legitimacy can be ensured by verifying if the ticket provided to the validator results from a legitimate transaction from a ticket box and that it isn't a unauthorized copy, forgery, already used or simply the wrong ticket. Valid ownership implies that there is some connection between the client and the ticket itself, meaning it shouldn't be usable by a third party.

Although some of these requirements are always true, context of use can be limiting as to the underlying security of the tickets. In paper storage, the most limiting, only a static key can be printed to it and any ownership validation would need to be done by hand. Smart cards are somewhat safer but anyone can use a card that does not belong to the legitimate user, therefore being a source of issues to the ownership check. Meanwhile on smartphone, assuming the phone data is protected, it would make it significantly harder to be used by a third-party. Also due to the smartphone's capabilities, it is possible to use dynamic keys for the tickets, further increasing security.

Another limit created by context comes from organizations themselves that might want tickets to be reused or to be transferable between users.

Chapter 4

Functional and Non-Functional Requirements

With the use cases defined, we can proceed to analyze them and determine the system requirements while also focusing on the planning as to the scope of this work. We can first state that we will define requirements for each of the actors of the e-ticket system: User, Service provider, and Validator. Doing this would allow us to, later on, define points of entry for the functionalities required for the satisfaction of the requirements. Since it is possible that some functionalities will not fall under the requirements of any of the aforementioned actors, these will be defined as under the purview of an Administrator to the system, in spite of such an actor not being considered in the system. Each of these points have their own requirements, however there are some important requirements that are shared amongst all actors. These would be mainly security related and it would involve the following:

- Secure end-to-end communication - Dealing with payments and providing a service, implies the use of proper security methods and maintaining a secure communication channel between the users and the server. Encrypting the messages is therefore essential along with secure information storage.
- Scalability - Although initially designed for small systems, proper scaling capabilities are required as to not to inhibit the growth of any user. Also, accepting the existence of multiple organizations to be registered under a single deployment, affects the scaling of the service, allowing the usage of the system by multiple simultaneous groups without disruption of service.
- Stability - Similar to other online services, disruptions to the service could imply a security or payment failure, which could in turn would make the system unusable. This requires the creation of fall-backs in case of any disruption occurs on any of the critical communications.
- Connectivity - Since development of the system's back-end is the scope of this work, a proper Application Programming Interface (API) implementation with the required documentation

is a must for any future work on a front-end interface that uses the API. Also, additional attention is required on the protocols or added third-parties as to avoid adding further software or hardware requirements to the front-end.

- Modular - Separation of services facilitates the systems stability and scalability, it also allows for easier maintenance and future improvement.
- Upgradability - Heavily affected by the modularity of how the system's architecture is designed and implemented, changes to protocols or additional service requirements are likely to happen. Maintaining the architecture adaptable to these changes allows for the utilization of less third-party software to handle the new requirements.

4.1 User requirements

These will affect functionality from the public's, or user's, side when using this system directly. Anything that involves the user planning to search, buying a ticket for and accessing an event has to be considered here.

- Login - A way to perform registration and login into the service, either by using a standard account or external registration (Social Media).
- Event search and display - Searching and listings of past, ongoing and future events and methods to filter the display. Afterwards it should give access to any relevant information about the event.
- Organization search and display - Similar to the above event search, but applied to organizations.
- Ticket acquisition - Methods to acquire the corresponding pretended ticket and display them for validation
- Ticket aggregation - Similar to acquisition of new tickets, however some cases require for the creating of a single verifiable package which includes multiple tickets.
- Secure payment - Some sort of payment system is required for any usage. In order to fulfill this requirement a communication module will be developed using the servers of MBWay mobile payment systems.
- Profile settings - Some functionality require a way to add and edit personal information, therefore a simple editable personal profile is needed.

4.2 Organization requirements

Some sort of back-office is needed so that organizations are able to edit their information, manage events, ticket categories, offers and their validators.

- Login - Secure log in using only local account information.
- Events information access - Access to any created event belonging to this organization.
- Event status - Some control over the state of the event, allowing the creation of new events without the need to publicize.
- Ticket categories - Categories to define different kind of tickets, along with a way to manage the price, sale periods and other relevant information
- Ticket offers - Ability and management of ticket offers allows for the system's native support of promotional actions, that the organization decides to undertake without a need for direct injection of the tickets.
- Create validation agents - Creating new validation agents involves the supply of a key to be used during the validator login.
- Validation agents lock - If for security reasons, it can required to invalidate a validator key, disabling any further logins or validation attempts.
- Payment config - Methods that govern how current payments are processed, therefore configuration is required although as stated above, initially only the MBWay payment system will be considered.
- Profile - Similar to the user, a simple editable profile is required.
- Access control - Although only a quality of life requirement, allowing for multiple accounts to administrate a single organization account adds some further security on the organization side.

4.3 Validator requirements

Validators are responsible for allowing access to events and catch any possible infraction. Therefore, the security and reliability of this component is crucial as any failure could either create a security breach or a considerable disruption of service.

- Login - A key should be given by organizations to their event validators to gain access to the validation API. This key should be of single use and locked onto the device on the first use to add control over the validating agents.

- Online Validation - During normal usage, agents are only a communication intermediary. They should only read the ticket information and transfer onto the server for processing. This processing should check the ticket's integrity and validity.
- Offline Validation - In case of any communication failure, validation cannot stop. Therefore, a means of offline validation is required even though it has reduced capabilities when compared to its online counterpart. This requirement, although considered as part of the project, should be facilitated and not performed by it.

4.4 Administrator requirements

These requirements are needed to perform more sensitive registrations. Everything that would fall on a direct to database access or in the need for a system administrator would fall under this category.

- Organization registration - Registering organizations should be initialized by service agents as to maintain some initial security. This registration involves the association of an organizational owner to a local account and creating the needed certificates for later on use in communication protocols.
- Ticket Seller certificate - For the same reasons as stated above, creating certificates to be used by third-parties ticket sellers should be done by administrative agents. This requirement only partially affects the current work, meaning the existence of these sources of tickets should already be considered by validators, however in testing phases, no actual third party is used.

Chapter 5

Architecture

In this chapter, having analyzed the system requirements stated above, we will plan the architecture of our system. First, it is possible to split the system into different services, henceforward called modules. Each module would satisfy the needs of a specific actor to the system and each section in this chapter will cover one of these modules. However an extra section exists, which will describe the auxiliary functionalities that serve more than one module.

This architectural design takes inspiration from modular programming, which allows independent development of each component of the software, and by running each of these modules as independent services, it maintains a clear separation of each of the actors, creating an obstacle to any unauthorized access to another component.

Common amongst all modules is the communication architecture. Being that the system should function independently of later developed end-user applications, we choose to implement the communication as a resource oriented web service, basing off Representational State Transfer (REST)[13] architecture which, with proper documentation allows for a strong and easy to use Application Programming Interface (API) without limiting any development choices on the application side. As for the communication, there was a choice between using the Simple Object Access Protocol (SOAP)[21] protocol or simple HyperText Transfer Protocol (HTTP) requests with JavaScript Object Notation (JSON)[10]. SOAP offers a very strict and formal messaging protocol which consequently, adds further security to the system, which is something critical as eventually, we would be processing payments. However, this protocol does not fit very well with REST architectures, since the API developed with the architecture is very mutable, as resources are added and accessed, while on SOAP, although not mandatory, not having static endpoints is a major obstacle for its implementation. Furthermore, the size of the messages produced by the protocol is substantial and with the added difficulty of parsing them, which would be a negative aspect considering the mobile context in which the system is planned to exist. On the other hand, JSON messages do not impart any added security by themselves, leaving this aspect to each end point so that they validate their own requests. However JSON objects are easily parsable and, unless required, should not contain any further information other than that which is strictly needed. This is a positive aspect for mobile development which, despite having

limited processing power, requires having fast interactivity with the user. Moreover, easier to parse messages facilitate the scalability of the system while maintaining fast responses.

Further similarities between the modules, are the existence of two main components, one of which will implement the REST architecture, validate the JSON requests, authenticate users, directing the body of the request into a second component and, after proper processing, respond to the request. Meanwhile, the second component will process the request and, as needed, communicate with the database to perform the access, creation or modifications on the resources. The last common component is the database. Although each module will be responsible for its own connection, they will all share the same database.

5.1 User

First, we will decide on the authentication methods. Initially, the planned authentication could be done using either of two methods: the user email, which would be verified, and user input password; or using a social network login. We will focus on the first method, since the external login requires only the implementation of the chosen network's API.

5.1.1 Local authentication

The email and password authentication, henceforward called local authentication, will be similar to most authentication protocols, although in this case, we will be creating an user Identification (ID) for internal processing using a SHA hash which is turn is digested into base 64 for storage. Furthermore, two other tokens are created using the same method. The first one is used later on for a time-based one-time key. The other one is used to identify a ticket storage container which, for the the context of this prototype, it would be the smartphone used during registration, however the definition of these containers is important for later support of paper and/or smart card storage mediums. The password will be handled using the bCrypt algorithm, salted with a random hash for later storage on a database.

After registration and as defined on the requirements, an user application would need endpoints in order to explore possible events or requested information, such as date, duration, ticket prices. Afterwards the main concern would be ticket acquisition, aggregation and payment.

Acquiring and aggregating the e-ticket can be abstracted into the same process, simply by dynamically creating packages, which can contain 1 or more tickets, and associating them to the user's container followed by the payment for that package.

5.1.2 Payment system

With the package formed, we should look at the payment. As proposed, the system should natively support communications with MBWay mobile payment systems, as such an interface is required.

But first, let's look at how MBWay works and its requirements. Communications to the MBWay servers are performed via a strict SOAP protocol, in which the webservice endpoints on MBWay's side determine what operation is performed. These operations can be financial operations, financial inquiries, alias control or TPA control. The first involves a request for payment or a request involving an earlier payment, for example a refund. Financial inquiries are used to retrieve information about payment history. Alias controls can be used to create an alias that refers to end users allowing the use of the returned token instead of any personal information on the end user and finally, the last possible operation is a request for the closure of the TPA. This last operation manually signals the end of the transactions for the current day allowing for the payment system to perform any required internal financial transactions.

Their API also requires the existence of endpoints on our side for asynchronous responses. More specifically, one endpoint is needed for asynchronous financial operation results and another for alias creation. However, we are not required to support all functionalities, therefore not having to satisfy all of MBWay's requirements. Instead we need only implement the requisites required for an early prototype of our system, more specifically the financial operations endpoints.

Knowing which communications will be required to perform, we can look at the protocol itself. Analyzing the MBWay technical documentation[6], we can assert that we will be playing the role of the integrator in one of two possible protocols. This signifies that, in a practical generic example, the merchant would register himself with us, giving their Point of Sale (POS) identification that they would be receiving from MBWay, which would allow us to perform payment requests on their behalf. Furthermore, we would need to register ourselves on MBWay, therefore acquiring a certificate to use on any requests performed to their endpoints. Additionally, for our asynchronous response endpoints, it is mandatory to use Hyper Text Transfer Protocol Secure (HTTPS) with our own certificates.

Taking in consideration how communications must be performed for the payment system, we know that a separate server listening for requests is needed along with a method to handle the asynchronous responses so that we can act accordingly. As such, a solution resides in the development of an additional independent module with its own database to log every incoming and outgoing request while maintaining the status of any pending requests.

Being that this module, henceforward referred as MBWay module, will be exchanging information with the user module, a means of communication to and from it is required, since we are already using HTTP for external requests, we decided to use the inter-process communication protocol, Inter-Process Communication (IPC), based on Unix sockets. Meaning that the MBWay module will be listening for events using an Unix socket to receive any required configurations

as well as to initiate any payment requests. These requests should already be arriving with all of the needed information and the module only works as a transparent middleman that logs information as needed. Increasingly, it will emit events for the asynchronous responses, to warn the user module of accepted, rejected or failed payments.

5.1.3 Ticket Usage

With the ticket acquired, the user's next step is accessing the ticket's token so that it can then be passed to the validator. The ticket is composed of two parts:

$$T = (K, S(K))$$

K is the key generated upon payment confirmation, and S(K) is K signed with the seller's certificate as specified in the section 5.4. In this work's prototype scope only one seller exists that being the system itself however, for future work, it is important to implement this component on the ticket. In section 5.3 we will analyze how the ticket works while undergoing validation.

5.2 Service Provider

5.2.1 Authentication and Access Control

To gain access onto this module, it is first required to create a local user account using the user module followed by registering an organization as explained on section 5.4. The initial account will be treated as the organization's owner and as such, it will have the permission to grant or remove access to other existing user accounts. These accounts should have access to most endpoints, except those that handle the authorized accounts and payment systems. As a security concern, no user account can have access to more than one organization.

5.2.2 Events creation and edition

Creating the event should require the input of core information, namely a name, description, start and end dates. Afterwards, it should be possible to further edit additional information. Out of these, the most noteworthy is the event state, image upload, ticket categories and validator control.

5.2.3 Event States

Some functionalities may depend upon the current situation of the event, for instance, if the event is already underway or yet to be publicized. Therefore it is required to define the current state of an event. As such, at any given time, the event will be in one of the following states:

- Saved - This will be the first state, which will be attributed right after creation. An event in this state won't have any information accessible by the user module.
- Planned - At any given time, the service provider may publicize this event transitioning it to the planned state. At this point, the information is accessible by the public and they should be able to acquire tickets.
- Active - During the time period in which the event is ongoing, it is considered as active.
- Finished - After the event conclusion time, it will be classified as finished.

5.2.4 Image Upload

Since the system should natively support any information and, although not critical to the core functionalities, a method to publish images for event publicizing would be a must for any practical implementation of this system. As such, and since our API will be based on JSON structures which is not ideal for file uploads, a point of entry to the upload of image files is required. This point of entry should have an encoding of *multipart/form-data* which allows files to be uploaded under a form. Afterwards, the request would return an identification for each uploaded file for later use in event edition.

5.2.5 Ticket Categories

Categories allow for a single event to have multiple tickets with different specifications. Each ticket category should have the following characteristics: name, description,price, amount, sale start time, sale end time, tags and secure.

Some of them are self explanatory, except the last two. Tags should work together with the validators adding the non-requirement functionality in which certain ones can only validate certain tickets. The second attribute determines if the ticket, when under online validation, requires the use of the time-based one-time key from the user, otherwise the ticket would be declared as invalid.

5.2.6 Validator Control

Along with the event and ticket management, service providers also need access to the means to manage the validators. This would require a method to create them, accessing their login tokens, attributing tags if using that feature and, for security reasons, be able to deactivate that token disabling any further online functionality.

5.2.7 Ticket Offers

Another non-requirement functionality that was added expecting to have a need for it later. The service provider may input emails in order to send to those emails an activation token for a given ticket package. Other than how these tickets are handed out, they function just the same as normally bought tickets.

5.3 Validator

5.3.1 Login

The token generated earlier is required to access this module. Furthermore, the system expects an application token, either generated by itself or using some identification from the smartphone, in order to lock the validator token to a device. This way the service provider could single out a device should any issues arise.

5.3.2 Validation

To perform the online validation, the validator should act as a transparent transporter of the ticket information from the user side application to the module. The endpoint should be expecting one of two possible JSON schemas. The first one contains only the ticket information referred on 5.1.3. In this case, we validate the ticket by first validating the signature and then checking the database for the current state of the ticket key. The second possibility, before looking at the ticket and the signature, we use the time-based one-time method.

At account creation, we generated a token and stored the timestamp of creation, which we proceed to share with the user. The time-based one-time algorithm[19] states that we use the HMAC-Based one-time password algorithm[18], however we define as such:

$$TOPT = HTOP(K, T)$$

, where K is used as the key and it is the token we shared earlier and T is the number of steps in the pre-set interval that have elapsed. In practice, we calculate T as

$$T = \frac{timestamp - current\_time}{interval}$$

.

5.4 Administrator

Currently, this module will only serve two purposes. The first is to register any new organizations on the system. For this it just requires the input of basic information such name, email, contact,

and description. It would also add the given user email owner status on the organization. The second purpose is to register third party ticket sellers. A certificate signing request[20] is required and from it we extract any needed information about the seller, we test the certificate, using it to test the signature on a message sent with the Certificate Signing Request (CSR) and if it passes we sign it followed by an email to the seller with the signed certificate.

5.5 Auxiliaries

These functionalities will be defined under a library which will be common to all modules. In this library, we will have implemented methods to perform:

- Logging - defines the different logging files and formats.
- Database connection - configurations, common points of entry for logging and interfaces to limit database queries.
- Emailing - Should implement a method to configure an email service and send any required emails
- JSON schema validation - Should contain regex or use a third-party library to validate communications into the server
- File Upload - Library to manage and control files being upload onto the server
- MBWay interface - This library should contain the required methods for communication with the aforementioned MBWay module.

Chapter 6

Implementation and Development

In this chapter we will look at how the aforementioned architecture is to be implemented, starting with the development of the Representational State Transfer (REST) Application Programming Interface (API), defining the database structure and then finally characterizing each of the modules and their functionalities. Some of the features that were added during the implementation were not intended to be fully functional by the end of this work. They are however intended to be basic ideas or structures in order to facilitate any future development. None of these features can affect the core functionality that was determined on the system requirements.

6.1 Database, Language choices and Environment

Before looking at the implementation, we should look at which tools are to be used.

Firstly, looking at the database, the first design choice to be made is between using a Structured Query Language (SQL) or a NoSQL database. Without going much deeper into the differences or advantages, it was decided to utilize a SQL database, simply due to the fact that it is the most commonly used on similar web-based services. As such, and due to prior experience with this technology, we have decided to use MySQL with the InnoDB engine.

The Next design choice to be made is to choose a coding language in which to implement the system. There are several to choose from along with frameworks designed for this purpose. In this case the decision made was to use JavaScript with the NodeJS framework. This decision was due to our familiarity in the use of this framework due to past projects. Furthermore, the existing NodeJS modules will accelerate the development and simplify the addition of features for future work.

Initially, we decided to make our implementation available on a remote server. For this purpose, a RaspberryPI server was used, unfortunately due to hardware failure we were forced to find an alternative. Looking at available alternatives that would allow us to keep developing

remotely we decided that renting a server would be our most viable option. However, due to the cost and performance loss over the RaspberryPI solution, we settled for using local virtual machines with Docker containers.

6.1.1 Docker configuration

We will be using Docker to simulate a network swarm with a single node. This would allow, in theory, for the future scaling of the system as needed to satisfy greater loads. However, in the scope of this work, the scalability was not fully tested. Nevertheless, a proper implementation of Docker would facilitate later work on that area.

As such, we will attribute to each module, as well as the database, their own service. Each service will be run in its own independent container over a lightweight Linux operating system. The configuration of each service will expose the required ports in the container to the host, create volumes that associate paths in the container to physical paths on the hosts, allowing for persistent data and a common point for the Inter-Process Communication (IPC) sockets. We will also introduce the required environment variables for any possible configuration as well as use the Docker secrets functionality, which stores any files containing sensitive data into "/run/secrets/", such as passwords and keys. The appendix A shows the full configuration used for this work.

6.2 Database Design

The design of the database was extended upon as different functionalities were being implemented as to satisfy their needs. We will now look into the structures that support the more critical functionalities.

Firstly, how to identify users. Initially we thought of using a simple user table, however as different log-in methods were considered, the following schema was decided upon, shown in figure 6.1, in which we are already considering the possibility of third party authentication.

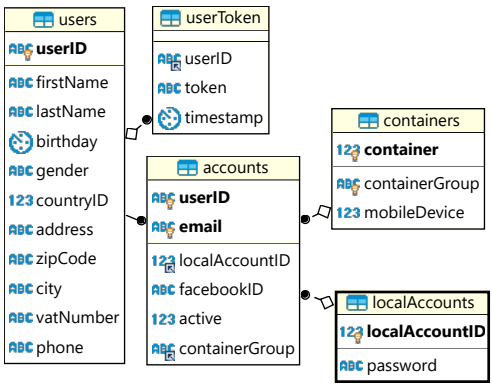


Figure 6.1: User RM Diagram

As can be seen, we use the table "users" to store personal information, this would have to

be reconsidered due to the General Data Protection Regulation that went into effect earlier this year. This also applies to some of the data stored in other tables. However, this work being a prototype and such information not being required by any functionality, this remains unaltered. The "accounts" table stores logistic information for authentication, including relations to tables where specific log-in methods' information is stored. The table "userToken" contains information required for the tickets' validation during the time-base one-time algorithm for each user. "containers" are, as the name suggests, used to relate possible multiple ticket containers to a single user, however during this prototype, each user will only have the possibility for one container, that will be identified as the mobile device.

For the service providers, as shown in figure 6.2, which were named "organizations", we show the relation from "users" to "organizationsAgents" which determines the users affiliated with the organization as well as the organization's owner. The "organizations" table itself contains mostly basic information. "mbway_organizations" is where we are storing the Point of Sale (POS) required for performing financial requests into the MBWay servers in the organization's behalf. Lastly, the figure shows one of the possible interactions of the "locations" table, which in addition to relating to any table where a location is required, is where Google Maps locations are stored.

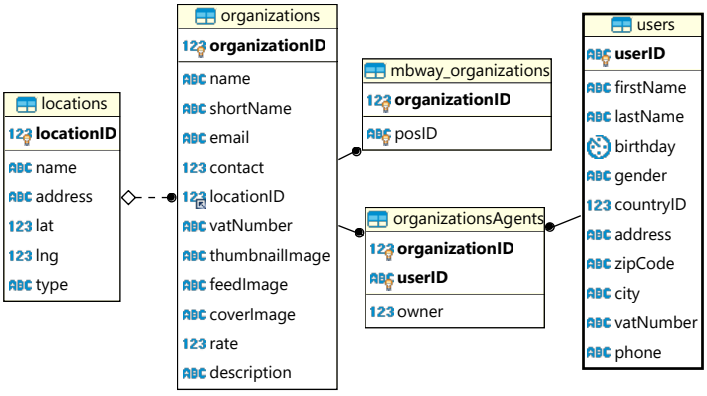


Figure 6.2: Service provider RM Diagram

In the figure 6.3 we show the relations between the event and its different components. Each event will start with an organization which owns the event and upon activation by the service provider, the entry is then added to "activeEvent", locking the event thus disabling its deletion. Furthermore on the service provider's request, entries can be added to "ticketTypes" or "verifiers". "ticketTypes" stores information as pertaining to the ticket categories, meanwhile, validators' information is stored in the table "verifiers". The relation between "verifiers" and "ticketTypes" can be used to determine which validators are authorized to validate certain tickets. The "eventRate" and "tagsEvents" were added as the need for rating and searching events by tags, for future work, was predicted.

Figure 6.4 shows which tables required to process and register a payment using MBWay. As can be seen, the table "mbway_transactions" registers all requested payments while maintaining an updated status. Each transaction should be related to a single ticket, an organization using the "posID" and a user using the "userID". We don't use the phone number associated with the

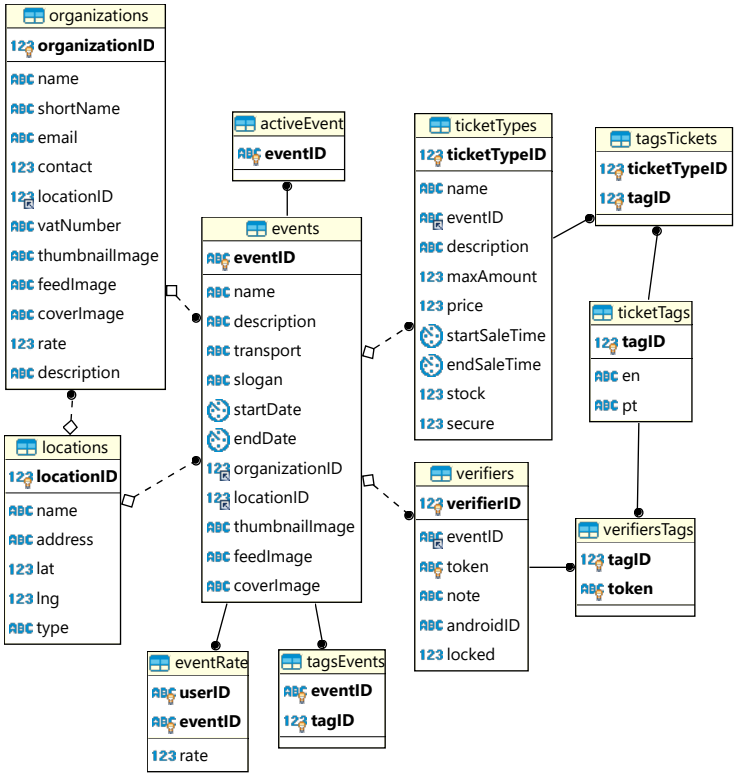


Figure 6.3: Event RM Diagram

MBWay directly since there is a possibility of it changing, furthermore, if required, it is easier to replace the table than to use an MBWay alias.

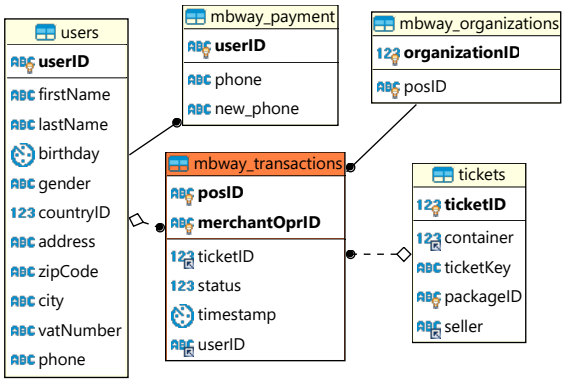


Figure 6.4: Financial Transactions RM Diagram

Lastly, figure 6.5, shows the "tickets" relation with the aforementioned "containers", the "ticket_package" relation between "ticketTypes" and "tickets" allowing for the creation of the required packages. Furthermore, we store the certificate corresponding to the ticket seller associated to a "boxOffice" and "invalidtickets" stores the Identification (ID)s of any ticket that were validated, refunded, cancelled or subject of payment failures.

Although we will not go into much detail, we also show the tables being used as logs for the MBWay module on figure 6.6. As can be observed, we keep a table for each outgoing request, one

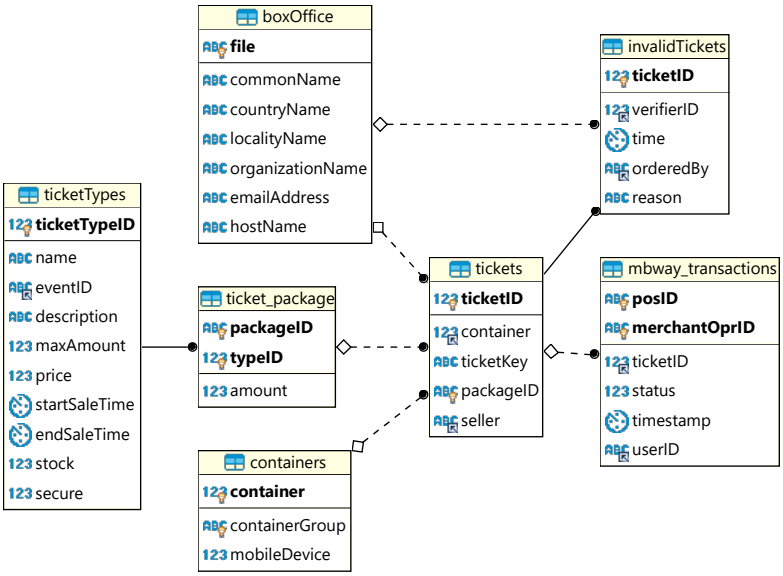


Figure 6.5: ER diagram with tables required for ticket creation and storage

registration annulment in case of communication errors, as per the MBWay specification; along with tables to store the synchronous and asynchronous responses with their respective status. Moreover, we also have tables for financial inquires, even though we will not be implementing that functionality on the system.

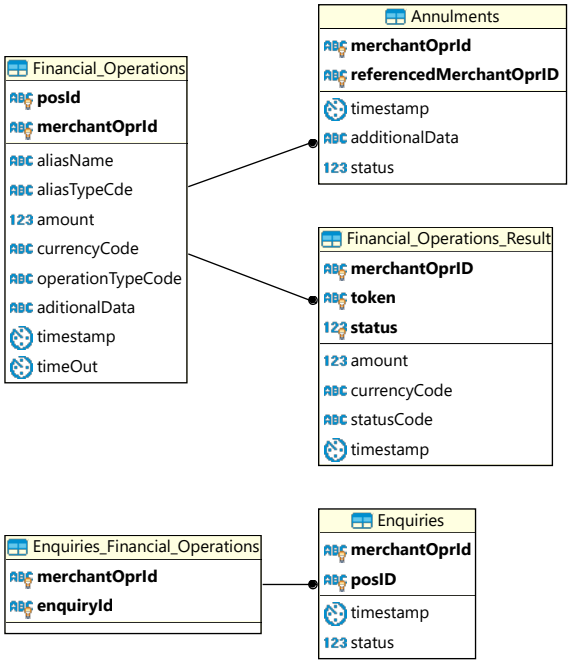


Figure 6.6: MBway Module RM Diagram

6.3 REST API

In order to follow the REST API's standard, we must define resources that will be created, edited, and deleted, as well as separating the function call entries from said resources, building them into adequate paths, and determining the most appropriate HyperText Transfer Protocol (HTTP) methods.

Looking at the modules and at the database design, we can state that only two modules, service provider and user, have resources whilst all of them will require endpoints that also act as function calls.

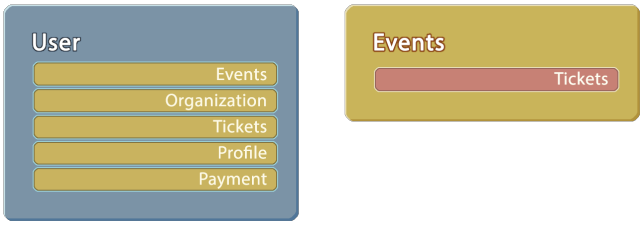


Figure 6.7: User resource diagram

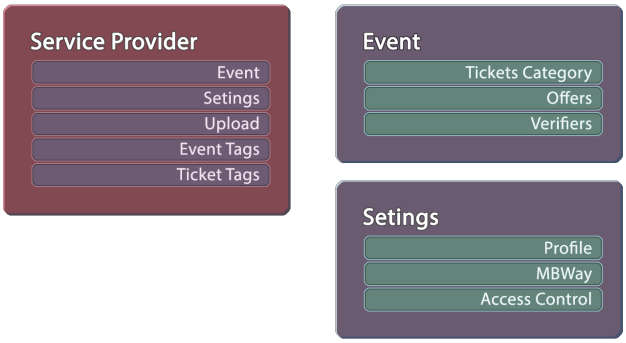


Figure 6.8: Service provider resource diagram

The diagrams 6.7 and 6.8 show the resources of each module. For these we will be using the following rules for paths nomenclature and methods:

- Paths - While building the path, we should use the following: /resource/resourceID;
- GET - Using this method will get a list of resources or, if provided with an ID, information on a specific one;
- POST - This method should be used to create new resources of the specified type;
- PUT - It should update the information of the resource. Unless otherwise noted by the API, it will always require the entire information of the resource, instead of just the changes;

- DELETE - As the name suggests, it should delete, if possible and allowed, the resource along with any child resources;

As for the function calls, they will cover authentication methods across all modules: the upload on the service provider, the ticket purchase on the user, and the ticket validation on the validator module. These calls will require a POST method, and both resources and function calls will use application/json content type except for the upload which, as stated before, will use multipart/form-data.

With this we start writing the documentation, using the OpenAPI 3.0 specification[7], for each of the API's modules, which are shown on appendix B.

6.4 Modules Implementation

In this section, we will discuss each of the individual modules that have been implemented using the NodeJS framework along with which, why, and how we choose certain third-party modules to tackle certain problems.

6.4.1 Logging

One of the advantages of using NodeJS is its middleware. This allows for the developer to process any request on a certain order depending on the paths. As such, we will use that potential as much as possible, starting with logging.

For the logging we will use a module called Winston, which allows for custom logging formats and outputs furthermore, if using Express-Winston we can intercept HTTP messages and log them.

To initialize, we will read the configuration used, with all the required information about transports and their formats being used on the module, along with, if required, adding the express application to the middleware to intercept the HTTP messages. Having initialized the logging, we only need to call it from any of the in-development modules that use either log or logErrors functions. It is of note that the log functions are designed to work asynchronously, while logErrors expects a callback chain that should end with the return of the proper error message to the user.

```
init: function (config) { ... }
initRoutingMiddleLogging: function (app) { ... }
logError: function (err, info, next) { ... }
log: function (level, txt) { ... }
```

Listing 6.1: Logging interface block

6.4.2 Database Access

We will use a module called MySQL to handle the database connection and queries. The configuration, which you can see in 6.2, is hard-coded since all modules should share the same database.

```
db_config.connectionLimit = 10;
db_config.dateStrings = true;
db_config.host = 'sql';
db_config.port = '3306';
db_config.user = 'tikt';
db_config.password = fs.readFileSync('/run/secrets/db_user_pass', 'utf8');
db_config.database = 'tikt';
db_config.multipleStatements = true;
```

Listing 6.2: Database configuration

The code block 6.3 shows the exposed interface. "ConnectDB" is used to initialize the connection, however it also catches connection errors or lost connections, in which case it will automatically attempt to reconnect after a certain amount of time has elapsed. The remaining functions do as their names suggest, being that "transactionError" also calls rollback on the current transactions and "getDataFromRows" returns extracts from the query response, in this case, an array with an object for each row.

```
function connectDB() { ... }
function procedure(procedureName, args, next) { ... }
function query(sql,values,next) { ... }
function beginTransaction(next) { ... }
function rollback(next) { ... }
function commit(next) { ... }
function transactionError(err, next) { ... }
function queryError(err, next) { ... }
function getDataFromRows(rows, next) { ... }
function format(sql, values) { ... }
```

Listing 6.3: Database interface

6.4.3 HTTPS Server

Each module will have its own Hyper Text Transfer Protocol Secure (HTTPS) server to implement and setup. The server will be using the express module, that natively allows for the use of middleware configurations from other modules. As such, during initialization we will introduce the middleware to be used on each of the modules, which can be seen on code block 6.4.

In order to use the HTTPS protocol we require the input of the Secure Sockets Layer (SSL)

```
app.use(cookieParser()); //Parse incoming cookies
app.use(bodyParser.json()); // Parse incoming body as application/json
app.use(session(config.cookies)); // Setup session storage with the cookie
    configuration
app.use(passport.initialize()); // Authentication module initialization
app.use(passport.session()); //  Setups the passport to use sessions
logger.initRoutingMiddleLogging(app); //  Starts the logging on HTTP requests
app.use('/', require('communication')); // Requests routing
```

Listing 6.4: Middleware initialization

certificate, key, and proper processing of the messages, however the module already takes care of it on the background as long we input the certificates. After that we can finally set the server to listen, as shown in the code block 6.5.

```
const privateKey = fs.readFileSync('/run/secrets/server.key', 'utf8');
const certificate = fs.readFileSync('/run/secrets/server.cert', 'utf8');
const credentials = {key: privateKey, cert: certificate};
const server = https.createServer(credentials, app);
server.listen(3001, () => { ... });
```

Listing 6.5: HTTPS initialization

6.4.4 Routing

Routing determines which functions are called for the path on the HTTP request. The router interface, as shown in the code block 6.6, parses the path as it moves from router to router with the 'use' function. It stops only when no further callback functions are possible, in this way we can represent the path names on the file-system folders.

```
router.use('/',authentication.auth,require('./general')); //  This method, if
    the current path corresponds to the argument, will call each of the
    functions or other routers in the argument progressively. If no path is
    given, it will call all the functions independently of the request.
router.all('/*', (req, res, next) => { ... } // This function will work on all
    HTTP methods as long as the path is correct.
router.route('/')
  .get((req, res) => { ... }
  .put((req, res) => { ... } // Same as above, but a method can be specified.
```

Listing 6.6: Router interface examples

Each route folder can be composed with the following files:

- index.js - The file that will be loaded with the 'use' function. It should contain the current folder's files and add paths for any sub-folder;
- routes.js - Contains the endpoints and methods to process the current path resource or function call;
- validations.js - Acts as middleware and it should be called by the endpoint. It should validate any user input on the path parameters, queries or body;
- headers.js - This file should be unique as it determines the headers configuration, mainly for cross-origin requests;

6.4.5 JSON validation

Validation for the incoming requests is performed as part of the routing. In each sub-folder, there is a "validation.js" file which requires the current module's "jsonSchemas.js" in order to validate all the incoming request bodies for the appropriate path, using the same router methods as those described above. The MBWay module will also use similar validation for incoming events from the user module.

6.4.6 Authentication

In order to implement the authentication, we resorted to a module called Passport, which allows for pre-built or custom authentication methods to be introduced as middleware. To call the proper methods we include the passport "authenticate" method within the routing, as exemplified in 6.7. These methods, that were loading during the initialization, should be described within the passport folder. The folder will contain the following files:

- index.js - Used to load all files under the folder and the folder itself is required;
- config.js - Contains the authentication methods mentioned above. On each method, we should validate user input and compare with the database as needed in order to return, or not, a user;
- passport_sql.js - Contains all the necessary queries required for this module's authentication;
- serializerconfig.js - Contains the serialize and de-serialize methods. They are used to write and read cookies;

6.4.7 Modules Root Folder Structure and Configurations

Each module will have a config.js file in the root. This file will contain the settings and format for the logging, the cookies for the configuration, and session storage. In the user module, it will also have the asynchronous handler for ticket status updates.

```
router.route('/login')
.post((req,res)=>{
  passport.authenticate('local-OrgLogin', (err, response) => { ... })(req);
});
```

Listing 6.7: Authentication in routing example

The module root should always have the same structure:

- config.js - Configuration file mention above;
- *moduleName*Module.js - Module executable. A procedure which will initialize everything;
- jsonSchemas.js - File with the schemas for validation on this module;
- communication - Folder for the routers;
- control - Folder with the files that include control methods. As stated earlier, every file in this folder is one of a pair. One file has the methods to be called by the router depending on the request and the other, identified by the "sql" suffix, contains the queries;
- passport - Folder with the module passport configuration;

6.4.8 Ticket Purchase

When the user requests the purchase of a ticket, a certain amount of steps are required:

- get POS - We require the registered POS id of the service provider in order to perform the request in their behalf;
- get seller certificate - Certificate of the ticket seller, however in current scope of this work, it will always be the certificate created for the ticket itself, since it is the only existing seller. This certificate is currently stored within the system;
- generate merchant operation id - unique ID required by MBWay to identify this operation;
- create package - With the user request ticket types, we create a new package definition;
- reserve tickets and register transaction - We are required to reduce the ticket stock, if possible, and register this transaction in the system's database;

6.4.9 MBWay Interface

We require a interface that implements the IPC events needed to communicate with the MBWay module and handle the asynchronous responses. First, to handle the responses, we must setup

a controller, host and port for it. The controller will be used by the interface to redirect the response back into the user module, while the host and port are used for setting up the MBWay module. For the IPC we also input some configuration, however the only the one required is the socket folder location.

```
const controller = require(process.env.MBWayController);
const host = "tikt.ddns.net";
const port = 3002;
ipc.config.appspace = 'tikt.';
ipc.config.retry = 5000;
ipc.config.socketRoot='/tmp/socket/';
```

Listing 6.8: MBWay interface configuration example

Next we need to setup the events that we will be emitted by and to the server. The events that will be emitted are:

- connect - automatically performed when IPC starts and finds the socket;
- init - after we receive the reply from connect, we emit this event with the configurations for the MBWay module;
- financeRequest - this event requires all the information for a financial transaction and when emitted it prompts the module to request the transaction;

As for the events we need to listen for:

- connect - Confirmation of connection;
- init_reply - Confirmation that module received configurations and has started;
- financeRequest_reply - Synchronous response from the MBWay servers. It is possible that we received an error or were told to wait for user input. Either way we update the database in case we were told to wait, otherwise we disable the ticket;
- financialAsync - asynchronous response from the MBway servers. We can be told that the user accepted or rejected the payment and the interface should enable or disable the ticket accordingly;
- timedOut - Event sent by module in case of any timed out happening.
- unexpected - Event sent by module in case of unexpected answers or internal errors.

During the process of requesting a financial transaction, as previously stated, we insert a registry of the transaction into the database. This registry should be updated as the process advances as described in the events. Furthermore, we also start a timer for an internal timed out control, which is disabled when we receive the "financialAsync" event.

```
function setcontroller(controller){ ... } \\ path for the controller file
function initiateTransaction(pos,merchantOprID,userID,ticketID,timestamp){ ...
    } \\Information required to initiate transaction
function closeConnection(){ ... } \\ closes IPC connection
```

Listing 6.9: MBWay interface

6.4.10 MBWay Module

Here the MBWay module will be discussed. The module is composed by four main components: the IPC interface to communicate with other modules, an HTTPS client to perform the initial requests, the HTTPS server to receive the asynchronous responses from MBWay, and a controller to process all the information. We will not describe the IPC since the configuration and events have already indirectly been described in subsection 6.4.9.

6.4.10.1 HTTPS client and server

The HTTPS client is used to make the requests as they are received on the IPC interface or, in certain cases, to independently emit a request for cancellation of the transaction. In either case, before emitting the request we create the appropriate XML envelope using the information stored in a JavaScript Object Notation (JSON) object.

For this, we have the requests template in a file with and a function to parse the information into the template. These files are under the 'xml' folder of this module. In this folder we can also find the files with methods to perform the reverse, which is parsing the responses from the MBWay servers back into JSON objects.

The client is also responsible to process the synchronous responses and convey them to the controller. Meanwhile the server, which proceeds in a similar way, listens for the asynchronous responses, parses them and directs them into the controller.

6.4.10.2 Controller

The controller will receive a JSON object corresponding to the information that was transmitted to the MBWay servers or received from them. Independently of its origin, the first thing to be done is creating an appropriate entry into the database, effectively logging all the communications. Then, it proceeds to emit the right event using the IPC interface. Also, in case of the synchronous positive response, it creates a timer to start an annulment process if no asynchronous response is received, as per the recommended procedure on the MBWay documentation.

6.4.11 Administrator Module

This module is, at the moment, responsible for only two things: processing certificate sign requests, and adding new service providers.

To add a new organization, we simply require a valid user email that is not associated to any other provider and base information about the service provider itself: name, short name, email, contact, VAT number, and description.

For the certificate signing requests we required the Certificate Signing Request (CSR) in the PKCS#10 specification[20], a message with any text and a hash which is the previous message signed with the key that pairs with the sent certificate. We then proceed to use a module called 'node-forge', which is also used for the tickets' signature, to extract information from the CSR, to validate the hash, and to sign the certificate before emailing it to the email contained within the certificate.

Chapter 7

Tests, Results, Problems

The prototype that would originate from this work was intended to be tested on a non-controlled environment by a real service provider during a real event. For this, parallel to its development, third-parties were working on solutions to work with the respective modules. However, due to circumstances beyond our control we were not able to achieve this goal.

Nevertheless, as proposed, a working prototype of the system with all core functionalities was developed and it has been in testing, not as intended, but by using Postman, a program which allows us to test Representational State Transfer (REST) Application Programming Interface (API)s, to simulate the applications.

The tests consisted on running several requests onto the server in which each test an individual functionality. Furthermore, when possible, the information used in a request would originate from a previous one. For example, before editing an event, we create an event, which gives us its Identification (ID), attempt to get its information with that ID and only then proceed to editing using the same ID. This would simulate a possibility of use of the API by an application.

To test the MBWay, we had entered in contact with the company to request access to their servers. Still early in development, we were given all documentation along with a certificate, key pair to communicate with the testing environment servers along with a testing application to fully simulate a real MBWay account. This allowed us to fully simulate a purchase with their services, including accept or deny payment requests as we were real users. However, near the end phases, of development problems occurred. Our access was cut off and we could no longer perform end-to-end tests.

Even though most of the functionality was already tested at this point, we still considered implementing an alternative testing method. As such, we resorted to the use of a MBWay stub server, whose configuration is provided by them and requires to use a program called SoapUI that creates stub servers with pre-programed responses.

7.0.1 Results

In order to show the results of the tests, we will enumerate each functionality on a table, followed by which requirement it falls under, or if it's an addition, and their success on the test.

Table 7.1: User module test results

Functionality	System Requirement	Result
Register	Login	Tested and working
Login	Login	Tested and working
Activate account	Login	Tested and working
Profile	Profile settings	Tested and working
Payment info	Profile settings	Tested and working
Get events	Event search and display	Tested and working
Get specific event	Event search and display	Tested and working
Get event tickets	Event search and display	Tested and working
Subscribe event	Addition	Tested however no method to fetch subscriptions
Unsubscribe event	Addition	Tested however no method to fetch subscriptions
Get organization	Organization search and display	Tested and working
Subscribe Organization	Addition	Tested however no method to fetch subscriptions
Unsubscribe Organization	Addition	Tested however no method to fetch subscriptions
Get valid tickets	Ticket acquirement	Tested and working
Get invalid tickets	Ticket acquirement	Tested and working
Validate a offer	Ticket acquirement	Tested and working
Purchase ticket	Ticket acquirement	Tested with stub and working

Table 7.2: Service provider module test results

Functionality	System Requirement	Result
Login	Login	Tested and working
Logout	Login	Tested and working
Event tags	Addition	Tested and working
Ticket category tags	Addition	Tested and working
Get all events	Events information access	Tested and working
Create event	Events information access	Tested and working
Get event	Events information access	Tested and working
Update event	Events information access	Tested and working
Delete saved event	Events information access	Tested and working
Upload files	Addition	Tested and working
Activate event	Event status	Tested and working
Create ticket category	Ticket categories	Tested and working
Get ticket category	Ticket categories	Tested and working
Update ticket category	Ticket categories	Tested and working
Delete ticket category	Ticket categories	Tested and working
Create validator	Create validation agents	Tested and working
Update validator	Validation agents lock	Tested and working
Get validator	Create validation agents	Tested and working
Create single offer	Ticket offers	Tested and working
Create bulk offer	Ticket offers	Tested and working however no method to distribute these bulk offers
Get offers	Ticket offers	Tested and working
Revoke bulk offer	Ticket offers	Tested and working
Revoke single offer	Ticket offers	Tested and working
Get profile	Profile	Tested and working
Update profile	Profile	Tested and working
Get authorized agents	Access control	Tested and working
Add authorized agent	Access control	Tested and working
Remove authorized agent	Access control	Tested and working
Get MBWay	Payment config	Tested and working
Update MBWay	Payment config	Tested and working

Table 7.3: Validator module test results

Functionality	System Requirement	Result
Login	Login	Tested and working
Validation without TOTP	Online Validation	Tested and working
Validation with TOTP	Online Validation	Tested and working
Access to certificates for partial offline validation	Offline Validation	Not implemented

Chapter 8

Conclusion and Future Work

Although we were able to satisfy all the proposed core functionalities and add some structure for future work, we were not able to test the system with the environment we wished. As such, we cannot say that the system is, at this point, as secure or that it can be considered a prototype ready for deployment, which was one of our main goals. Furthermore the system as it is, will still require some configuration in order to work properly.

Some of this deficiencies are, in part, due to lack of applications on which to test. With those applications and with more extensive testing, we could have found more points of failure or security breaches.

8.0.1 Future Work

We would propose that the next step is to apply the current prototype on the original planned testing environment. This would give a more clear idea of which functionalities, currently implemented, would require more work or are prone to failure in uncontrolled environments. For this, it would require the development of applications to interact with the system.

Having proceeded with those tests and after proper modifications, it would be possible to start implementing the open functionalities that were not planned for this version of the prototype such as event searching, tags filters, third-party ticket seller Application Programming Interface (API), applications for paper and smart card alternative storage,number verification for MBWay payments, and additional payment methods.

Appendix A

Docker Configuration

```
version: '3.2'
services:
  sql:
    image: mysql:5.7.22
    hostname: sql
    ports:
      - 3306:3306
    volumes:
      - ./db_data:/var/lib/mysql
      - ./db_config:/etc/mysql/conf.d
      - ./sqlinit:/docker-entrypoint-initdb.d
    secrets:
      - db_root_pass
      - db_user_pass
    environment:
      MYSQL_USER: tikt
      MYSQL_PASSWORD_FILE: /run/secrets/db_user_pass
      MYSQL_ROOT_PASSWORD_FILE: /run/secrets/db_root_pass
  provider:
    image: tikt:provider-dev
    hostname: provider
    ports:
      - 3001:3001
      - 9221:9229
    secrets:
      - server.key
      - server.cert
      - db_user_pass
    volumes:
      - ./public:/var/public
      - ./logs/provider:/var/log
      - type: tmpfs
        target: /tmp/sessions
    environment:
     - NODE_ENV=debug-windows
```

```
    command: node --inspect=0.0.0.0:9229 serviceProviderModule.js
  user:
    image: tikt:user-dev
    hostname: user
    ports:
      - 3002:3002
      - 9222:9229
    secrets:
      - server.key
      - server.cert
      - db_user_pass
      - TIKTBoxOffice.key
    volumes:
      - ./logs/user:/var/log
      - MBWaysocket:/tmp/socket/
      - ./boxOffice:/ssl/boxOffice
      - type: tmpfs
        target: /tmp/sessions
    environment:
     - NODE_ENV=debug-windows
    command: node --inspect=0.0.0.0:9229 userModule.js
  verifier:
    image: tikt:verifier-dev
    hostname: user
    ports:
      - 3003:3003
      - 9223:9229
    secrets:
      - server.key
      - server.cert
      - db_user_pass
    volumes:
      - ./boxOffice:/ssl/boxOffice
      - ./logs/verifier:/var/log
      - type: tmpfs
        target: /tmp/sessions
    environment:
     - NODE_ENV=debug-windows
    command: node --inspect=0.0.0.0:9229 verifierModule.js
  admin:
    image: tikt:admin-dev
    hostname: admin
    ports:
      - 3005:3005
      - 9235:9229
    secrets:
      - db_user_pass
      - TIKTBoxOffice.key
      - CA.key
      - CA.crt
    volumes:
```

```
      - ./logs/admin:/var/log
      - ./boxOffice:/ssl/boxOffice
    environment:
      - NODE_ENV=debug-windows
    command: node --inspect=0.0.0.0:9229 adminModule.js
  mbway:
    image: tikt:mbway-dev
    hostname: mbway
    ports:
      - 3004:3004
      - 9224:9229
    secrets:
      - server.key
      - server.cert
      - db_user_pass
      - SIBS_key
      - SIBS_pfx
    volumes:
      - MBWaysocket:/tmp/socket/
      - ./logs/mbway:/var/log
    environment:
      - NODE_ENV=debug-windows
    command: node --inspect=0.0.0.0:9229 MBwayHandler.js
volumes:
  MBWaysocket:
secrets:
  server.key:
    external: true
  server.cert:
    external: true
  db_root_pass:
    external: true
  db_user_pass:
    external: true
  SIBS_key:
    external: true
  SIBS_pfx:
    external: true
  TIKTBoxOffice.key:
    external: true
  CA.crt:
    external: true
  CA.key:
    external: true
```

Listing A.1: Docker configuration file

Appendix B

REST API

User API

GET /tickets/active

shows active tickets of the client (**clientActiveTickets**)

Return type
array[ticketInfo]

Example data
Content-Type: application/json

```
{
  "ticketKey" : "ticketKey",
  "ticket" : {
    "signature" : "signature",
    "key" : "key"
  },
  "packgeID" : "packgeID",
  "name" : "name",
  "thumbnailImage" : "thumbnailImage",
  "startDate" : "2000-01-23T04:56:07.000+00:00"
}
```

Produces
This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.

- `application/json`

Responses
200 success
401 Unauthorized

POST /events/{eventID}/buy

initiate ticket purchase (**clientBuyTicket**)

Path parameters
eventID (required)
Path Parameter — Event ID

Consumes
This API call consumes the following media types via the Content-Type request header:

- `application/json`

Request body
integer array (required)
Body Parameter —

Responses
200 success
400 something
401 Unauthorized

GET /events/{eventID}

grabs event information (**clientEvent**)

Path parameters
eventID (required)
Path Parameter — Event ID

Query parameters
lang (required)
Query Parameter — language to retrieve from tags table

Return type
event

Example data
Content-Type: application/json


```
{
  "organizationID" : "organizationID",
  "subscribed" : 2,
  "endDate" : "2000-01-23T04:56:07.000+00:00",
  "rate" : 5,
  "coverImage" : "coverImage",
  "organization" : {
    "organizationID" : "organizationID",
    "name" : "name",
    "shortName" : "shortName",
    "thumbnailImage" : "thumbnailImage"
  },
  "name" : "name",
  "description" : "description",
  "location" : {
    "address" : "address",
    "lng" : 1.46581298050294517310021547018550336360931396484375,
    "locationID" : 0,
    "name" : "name",
    "type" : "type",
    "lat" : 6.027456183070403206158971443073824048042297363281 25
  },
  "transport" : "transport",
  "startDate" : "2000-01-23T04:56:07.000+00:00",
  "tags" : [ {
    "tagid" : 5,
    "tag" : "tag"
  }, {
    "tagid" : 5,
    "tag" : "tag"
  } ]
}
```


Produces
This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.

- `application/json`

Responses
200 success event
401 Unauthorized

`GET /events/{eventID}/feed`
get event feed (**clientEventFeed**)

Path parameters
eventID (required)
Path Parameter — Event ID

Query parameters
startAt (required)
Query Parameter — start getting feed indexed at given int (inclusive) format: int32

Return type
feedArray

Example data
Content-Type: application/json


```
{
```

```
  "feed" : {
    "eventID" : "eventID",
    "image" : "image",
    "feedID" : 0,
    "publishDate" : "publishDate",
    "text" : "text"
  }
}
```

Produces
This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.

- `application/json`

Responses
200 success feedArray
401 Unauthorized

GET /events/{eventID}/tickets

grabs all ticket information from the event (**clientEventTickets**)

Path parameters
eventID (required)
Path Parameter — Event ID

Return type
array[eventTicket]

Example data
Content-Type: application/json

```
{
  "hasStock" : 1,
  "price" : 0,
  "name" : 6,
  "description" : "description",
  "ticketTypeID" : "ticketTypeID"
}
```

Produces
This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.

- `application/json`

Responses
200 success
401 Unauthorized

GET /feed

Gets all feed information (**clientFeed**)

Query parameters
startAt (required)
Query Parameter — start getting feed indexed at given int (inclusive)

Return type
feedArray

Example data
Content-Type: application/json

```
{
  "feed" : {
    "eventID" : "eventID",
    "image" : "image",
    "feedID" : 0,
    "publishDate" : "publishDate",
    "text" : "text"
```

```
  }
}
```

Produces
This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.

- `application/json`

Responses
200	success feedArray
400	bad parameter
401	Unauthorized

GET /events
Gets all events (**clientGetEvents**)

Return type
events

Example data
Content-Type: application/json

```
{
  "active" : [ {
    "eventID" : "eventID",
    "name" : "name",
    "thumbnailImage" : "thumbnailImage",
    "startDate" : "2000-01-23T04:56:07.000+00:00"
  }, {
    "eventID" : "eventID",
    "name" : "name",
    "thumbnailImage" : "thumbnailImage",
    "startDate" : "2000-01-23T04:56:07.000+00:00"
  } ],
  "planned" : [ {
    "eventID" : "eventID",
    "name" : "name",
    "thumbnailImage" : "thumbnailImage",
    "startDate" : "2000-01-23T04:56:07.000+00:00"
  }, {
    "eventID" : "eventID",
    "name" : "name",
    "thumbnailImage" : "thumbnailImage",
    "startDate" : "2000-01-23T04:56:07.000+00:00"
  } ]
}
```

Produces
This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.

- `application/json`

Responses
200	success events
401	Unauthorized

GET /payment
shows user payment (**clientGetPayment**)

Return type
payment

Example data
Content-Type: application/json

```
{
  "mbway" : {
    "phone" : "phone"
```

```
}
```

Produces
This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.
- `application/json`

Responses
200 success payment
401 Unauthorized

GET /profile
shows user profile (**clientGetProfile**)

Return type
profile

Example data
Content-Type: application/json

```
{
  "something" : "something"
}
```

Produces
This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.
- `application/json`

Responses
200 success profile
401 Unauthorized

GET /tickets/inactive
shows innactive tickets of the client (**clientInnactiveTickets**)

Return type
array[invalidTicket]

Example data
Content-Type: application/json

```
{
  "reason" : "reason",
  "thumbailImage" : "thumbailImage",
  "name" : "name",
  "time" : "2000-01-23T04:56:07.000+00:00",
  "startDate" : "2000-01-23T04:56:07.000+00:00"
}
```

Produces
This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.
- `application/json`

Responses
200 success
401 Unauthorized

POST /local/sendEmail
Sends user validtion email (**clientLocalEmail**)

Consumes
This API call consumes the following media types via the Content-Type request header:
- `application/json`

Request body
body **string** **(required)**
Body Parameter —

Responses
200 email sent
400 bad input parameter

POST /local/login
Logins into client platform (**clientLocalLogin**)

Consumes
This API call consumes the following media types via the Content-Type request header:
- `application/json`

Request body
login **login** **(required)**
Body Parameter —

Return type
loginInfo

Example data
Content-Type: application/json

```
{
  "userID" : "userID"
}
```

Produces
This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.
- `application/json`

Responses
200 sucesseful login loginInfo
400 bad input parameter
401 bad login information or user not active

POST /local/signup
signup into client platform (**clientLocalSignup**)

Consumes
This API call consumes the following media types via the Content-Type request header:
- `application/json`

Request body
localSignup **localSignup** **(required)**
Body Parameter —

Return type
inline_response_200

Example data
Content-Type: application/json

```
{
  "userID" : "userID",
  "email" : "email",
  "token" : "token",
  "timestamp" : "timestamp"
}
```

Produces
This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.
- `application/json`

Responses
200 sucesseful signup inline_response_200
400 bad input parameter

GET /local/sendEmail

validate client's email (**clientLocalValidation**)

Path parameters

tempToken (required)
Path Parameter — temporary token

Responses

200 sucesseful signup
400 bad input parameter

POST /logout

Logout of the platform. (**clientLogout**)

Responses

200 sucesseful logout

GET /org/{orgID}

get organization info (**clientOrg**)

Path parameters

orgID (required)
Path Parameter — Organization ID

Return type

organization

Example data

Content-Type: application/json


```
{
  "finishedEvents" : [ {
    "eventID" : "eventID",
    "name" : "name",
    "thumbnailImage" : "thumbnailImage",
    "startDate" : "2000-01-23T04:56:07.000+00:00"
  }, {
    "eventID" : "eventID",
    "name" : "name",
    "thumbnailImage" : "thumbnailImage",
    "startDate" : "2000-01-23T04:56:07.000+00:00"
  } ],
  "subscribed" : 6,
  "rate" : 0,
  "contact" : "contact",
  "coverImage" : "coverImage",
  "name" : "name",
  "plannedEvents" : [ {
    "eventID" : "eventID",
    "name" : "name",
    "thumbnailImage" : "thumbnailImage",
    "startDate" : "2000-01-23T04:56:07.000+00:00"
  }, {
    "eventID" : "eventID",
    "name" : "name",
    "thumbnailImage" : "thumbnailImage",
    "startDate" : "2000-01-23T04:56:07.000+00:00"
  } ],
  "description" : "description",
  "activeEvents" : [ {
    "eventID" : "eventID",
    "name" : "name",
    "thumbnailImage" : "thumbnailImage",
    "startDate" : "2000-01-23T04:56:07.000+00:00"
```

```
  }, {
    "eventID" : "eventID",
    "name" : "name",
    "thumbnailImage" : "thumbnailImage",
    "startDate" : "2000-01-23T04:56:07.000+00:00"
  } ],
  "location" : {
    "address" : "address",
    "lng" : 1.4658129805029451731002154701855033636093139648437 5,
    "locationID" : 0,
    "name" : "name",
    "type" : "type",
    "lat" : 6.027456183070403206158971443073824048042297363281 25
  },
  "shortName" : "shortName",
  "email" : "email"
}
```


Produces
This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.
- `application/json`

Responses
200	success [organization](#)
401	Unauthorized

POST /event/{eventID}/rate
gives rating to event (**clientRate**)

Path parameters
eventID (required)
Path Parameter — Event ID

Consumes
This API call consumes the following media types via the Content-Type request header:
- `application/json`

Request body
eventRate [eventRate](#) (required)
Body Parameter —

Responses
200	success
400	unable to rate
401	Unauthorized

GET /search
search given name in events and organizations (**clientSearch**)

Query parameters
name (required)
Query Parameter — name to search

Return type
[searchResult](#)

Example data
Content-Type: application/json


```
{
  "organizations" : [ {
    "organizationID" : "organizationID",
    "name" : "name",
    "shortName" : "shortName",
    "thumbnailImage" : "thumbnailImage"
  }, {
    "organizationID" : "organizationID",
    "name" : "name",
```

```
    "shortName" : "shortName",
    "thumbnailImage" : "thumbnailImage"
  } ],
  "events" : [ {
    "eventID" : "eventID",
    "name" : "name",
    "thumbnailImage" : "thumbnailImage",
    "startDate" : "2000-01-23T04:56:07.000+00:00"
  }, {
    "eventID" : "eventID",
    "name" : "name",
    "thumbnailImage" : "thumbnailImage",
    "startDate" : "2000-01-23T04:56:07.000+00:00"
  } ]
}
```


Produces

This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.

- `application/json`

Responses

200	success searchResult
401	Unauthorized

POST /events/{eventID}/subscribe

subscribe to event feed (**clientSubscribeEvent**)

Path parameters

eventID (required)
Path Parameter — Event ID

Responses

200	success
400	unable to subscribe
401	Unauthorized

POST /org/{orgID}/subscribe

subscribe to organization feed (**clientSubscribeOrg**)

Path parameters

orgID (required)
Path Parameter — Organization ID

Responses

200	success
400	unable to subscribe
401	Unauthorized

DELETE /events/{eventID}/subscribe

unsubscribe to event feed (**clientUnsubscribeEvent**)

Path parameters

eventID (required)
Path Parameter — Event ID

Responses

200	success
400	unable to unsubscribe
401	Unauthorized

DELETE /org/{orgID}/subscribe

unsubscribe to organization feed (**clientUnsubscribeOrg**)

Path parameters

orgID (required)
Path Parameter — Organization ID

Responses
200 success
400 unable to unsubscribe
401 Unauthorized

POST /payment
updates user payment (**clientUpdatePayment**)

Consumes
This API call consumes the following media types via the Content-Type request header:
- `application/json`

Request body
payment **payment** **(required)**
Body Parameter —

Responses
200 success
401 Unauthorized

PATCH /profile
updates user profile (**clientUpdateProfile**)

Consumes
This API call consumes the following media types via the Content-Type request header:
- `application/json`

Request body
profile **profile** **(required)**
Body Parameter —

Responses
200 success
401 Unauthorized

Models

eventSummary -
name
String

thumbnailImage
String

startDate
Date format: date-time

eventID
byte[] format: byte

eventTicket -
ticketTypeID
byte[] format: byte

price
Integer format: int32

name
Integer format: int32

hasStock
Integer format: int32

description
String

inline_response_200 -
token (optional)
String

timestamp (optional)
String

userID (optional)
String

email (optional)
String

invalidTicket -
name
String

thumbailImage (optional)
String

startDate
Date format: date-time

time
Date format: date-time

reason
String

event -
- name
  String
- description
  String
- transport
  String
- startDate
  Date format: date-time
- endDate
  Date format: date-time
- location
  location
- rate
  Integer format: int32
- coverImage
  String
- organizationID
  byte[] format: byte
- tags
  array[tag]
- organization
  organizationSummary
- subscribed
  Integer format: int32

eventRate -
- rate
  Integer format: int32

events -
- active
  array[eventSummary]
- planned
  array[eventSummary]

feed -
- eventID
  byte[] format: byte
- text
  String
- image
  String
- feedID
  Integer format: int32
- publishDate
  String

feedArray -
- feed
  feed

localSignup -
- email
  String format: email
- password
  String

location -
- locationID
  Integer format: int32
- name
  String
- address
  String
- lat
  BigDecimal
- lng
  BigDecimal
- type
  String

login -
- email
  String format: email
- password
  String
- rememberMe
  Boolean

loginInfo -
- userID
  byte[] format: byte

organizationSummary -
- name
  String
- shortName
  String
- thumbnailImage
  String
- organizationID
  byte[] format: byte

payment -
- mbway
  payment_mbway

payment_mbway -
- phone (optional)
  String

profile -
- something
  String

searchResult -
- events
  array[eventSummary]
- organizations
  array[organizationSummary]

tag -
- tagid
  Integer format: int32
- tag
  String

organization -

name
String

shortName
String

description
String

location
location

email
String format: email

contact
String

coverImage
String

activeEvents
array[eventSummary]

plannedEvents
array[eventSummary]

finishedEvents
array[eventSummary]

rate
Integer format: int32

subscribed
Integer format: int32

ticket -

key
String

signature
String

ticketInfo -

name
String

startDate
Date format: date-time

thumbnailImage
String

packgeID (optional)
byte[] format: byte

packageInfo
ticketPackage

ticketKey
String

ticket (optional)
ticket

ticketPackage -

ticketPackage_inner -

amount (optional)
Integer format: int32

typeID (optional)
Integer format: int32

name (optional)
String

description (optional)
String

types -

Service Prorivder API

POST /event/{eventID}/activate

activate an event (**orgAcvtivateEvent**)

Path parameters

eventID (required)
Path Parameter — Event ID

Responses

200	success
400	Activation not avaiable
401	Unauthorized

POST /event/{eventID}/offers

add offer to event (**orgAddOffers**)

Path parameters

eventID (required)
Path Parameter — Event ID

Consumes

This API call consumes the following media types via the Content-Type request header:

- `application/json`

Request body

newOffer newOffer **(required)**
Body Parameter —

Responses

201	success
400	bad input parameter
401	Unauthorized

POST /event/{eventID}/verifiers

add new verifier (**orgAddVerifier**)

Path parameters

eventID (required)
Path Parameter — Event ID

Consumes

This API call consumes the following media types via the Content-Type request header:

- `application/json`

Request body

newVerifier newVerifier **(required)**
Body Parameter —

Responses

201	success
400	bad input parameter
401	Unauthorized

GET /event/{eventID}/verifiers/{verifierToken}

Get verifier info (**orgAddVerifier_1**)

Path parameters

eventID (required)
Path Parameter — Event ID

verifierToken (required)
Path Parameter — verifier token

Return type

verifierInfo

Example data

Content-Type: application/json

```
{
  "note" : "note",
  "lock" : true,
  "tags" : [ 0, 0 ]
}
```

Produces
This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.
- `application/json`

Responses
200	success verifierInfo
401	Unauthorized

DELETE /event/{eventID}
delete event (**orgDeleteEvent**)

Path parameters
eventID (required)
Path Parameter — Event ID

Responses
200	success
400	Deletion not avaiable
401	Unauthorized

DELETE /event/{eventID}/ticketType/{ticketTypeID}
delete ticket type (**orgDeleteTicketType**)

Path parameters
eventID (required)
Path Parameter — Event ID

ticketTypeID (required)
Path Parameter — Organization ID

Responses
200	success
400	Deletion not avaiable
401	Unauthorized

GET /eventTags
Get tags relative to events (**orgEventTags**)

Query parameters
lang (required)
Query Parameter — language to retrieve from tags table

Return type
array[tagArray]

Example data
Content-Type: application/json

```
{
  "id" : 0,
  "tag" : "tag"
}
```

Produces
This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.
- `application/json`

Responses
200	returning event tag information
400	bad input parameter

401 Unauthorized

GET /settings/auth

get organization authorized accounts (**orgGetAuth**)

Return type
array[String]

Example data
Content-Type: application/json

```
null
```

Produces
This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.
- `application/json`

Responses
200 success
401 Unauthorized

GET /event/{eventID}

get base information of an event (**orgGetEventInfo**)

Path parameters
eventID (required)
Path Parameter — Event ID

Return type
eventInfo

Example data
Content-Type: application/json


```
{
  "endDate" : "2000-01-23T04:56:07.000+00:00",
  "active" : true,
  "description" : "description",
  "transport" : "transport",
  "tags" : [ 5, 5 ],
  "feedImage" : "feedImage",
  "coverImage" : "coverImage",
  "name" : "name",
  "ticketTypes" : [ {
    "name" : "name",
    "description" : "description",
    "ticketTypeID" : "ticketTypeID",
    "stock" : 5
  }, {
    "name" : "name",
    "description" : "description",
    "ticketTypeID" : "ticketTypeID",
    "stock" : 5
  } ],
  "location" : {
    "address" : "address",
    "lng" : 1.46581298050294517310021547018550336360931396484375,
    "locationID" : 0,
    "name" : "name",
    "type" : "type",
    "lat" : 6.027456183070403206158971443073824048042297363281 25
  },
  "thumbnailImage" : "thumbnailImage",
  "slogan" : "slogan",
  "startDate" : "2000-01-23T04:56:07.000+00:00",
  "verifiers" : [ {
```

```
    "note" : "note",
    "locked" : true,
    "token" : "token"
  }, {
    "note" : "note",
    "locked" : true,
    "token" : "token"
  } ]
}
```


Produces
This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.
- `application/json`

Responses
200 success eventInfo
401 Unauthorized

Up

`GET /event`
get all events of the organization (**orgGetEvents**)

Return type
events

Example data
Content-Type: application/json


```
{
  "active" : [ {
    "eventID" : "eventID",
    "name" : "name",
    "thumbnailImage" : "thumbnailImage",
    "startDate" : "2000-01-23T04:56:07.000+00:00"
  }, {
    "eventID" : "eventID",
    "name" : "name",
    "thumbnailImage" : "thumbnailImage",
    "startDate" : "2000-01-23T04:56:07.000+00:00"
  } ],
  "planned" : [ {
    "eventID" : "eventID",
    "name" : "name",
    "thumbnailImage" : "thumbnailImage",
    "startDate" : "2000-01-23T04:56:07.000+00:00"
  }, {
    "eventID" : "eventID",
    "name" : "name",
    "thumbnailImage" : "thumbnailImage",
    "startDate" : "2000-01-23T04:56:07.000+00:00"
  } ]
}
```


Produces
This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.
- `application/json`

Responses
200 success events
401 Unauthorized

Up

`GET /feed`
get organization feed (**orgGetFeed**)

Query parameters
startAt (required)
Query Parameter — start getting feed indexed at given int (inclusive) format: int32

```
{
  "feed" : {
    "eventID" : "eventID",
    "image" : "image",
    "feedID" : 0,
    "publishDate" : "publishDate",
    "text" : "text"
  }
}
```

Produces
This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.
- `application/json`

Responses
200 success feedArray
401 Unauthorized

GET /feed/{feedID}
get organization feed of a event (**orgGetFeed_2**)

Path parameters
feedID (required)
Path Parameter — Feed ID

Return type
feed

Example data
Content-Type: application/json

```
{
  "eventID" : "eventID",
  "image" : "image",
  "feedID" : 0,
  "publishDate" : "publishDate",
  "text" : "text"
}
```

Produces
This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.
- `application/json`

Responses
200 success feed
401 Unauthorized

GET /settings/mbway
get organization mbway payment information (**orgGetMBWay**)

Return type
mbway

Example data
Content-Type: application/json

```
{
  "posID" : "posID"
}
```

Produces
This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.
- `application/json`

Responses
200 success mbway
401 Unauthorized

GET /event/{eventID}/offers
get event offers (**orgGetOffers**)

Path parameters
eventID (required)
Path Parameter — Event ID

Return type
offers

Example data
Content-Type: application/json

```
{
  "single" : [ {
    "origin" : "origin",
    "packageID" : "packageID",
    "offerID" : "offerID",
    "email" : "email",
    "status" : 0
  }, {
    "origin" : "origin",
    "packageID" : "packageID",
    "offerID" : "offerID",
    "email" : "email",
    "status" : 0
  } ],
  "bulk" : [ {
    "amount" : 6,
    "packageID" : "packageID",
    "offerID" : "offerID",
    "stock" : 1,
    "email" : "email"
  }, {
    "amount" : 6,
    "packageID" : "packageID",
    "offerID" : "offerID",
    "stock" : 1,
    "email" : "email"
  } ]
}
```

Produces
This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.
- `application/json`

Responses
200 success offers
401 Unauthorized

GET /settings/profile
get organization profile info (**orgGetProfile**)

Return type
profile

Example data
Content-Type: application/json

```
{
  "feedImage" : "feedImage",
  "contact" : "contact",
```

```
  "coverImage" : "coverImage",
  "thumbailImage" : "thumbailImage",
  "name" : "name",
  "location" : "location",
  "shortName" : "shortName",
  "email" : "email"
}
```

Produces
This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.
- `application/json`

Responses
200 success [profile](#)
401 Unauthorized

GET /settings/mbway/transations
get organization transations (STUB) (**orgGetTransation**)

Responses
200 success
401 Unauthorized

GET /event/{eventID}/ticketType/{ticketTypeID}
get ticket type information (**orgGetType**)

Path parameters
eventID (required)
Path Parameter — Event ID

ticketTypeID (required)
Path Parameter — Organization ID

Return type
[ticketType](#)

Example data
Content-Type: application/json

```
{
  "amount" : 6,
  "price" : 0,
  "name" : "name",
  "description" : "description",
  "endSaleTime" : "2000-01-23T04:56:07.000+00:00",
  "stock" : 1,
  "secure" : true,
  "startSaleTime" : "2000-01-23T04:56:07.000+00:00",
  "tags" : [ 5, 5 ]
}
```

Produces
This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.
- `application/json`

Responses
200 success [ticketType](#)
401 Unauthorized

GET /home
TODO would return to main page (**orgHome**)

Responses
200 success
401 Unauthorized

POST /login

Logins into organization platform. (**orgLogin**)

Consumes

This API call consumes the following media types via the Content-Type request header:

- `application/json`

Request body

login **login** **(required)**
Body Parameter —

Return type

loginInfo

Example data

Content-Type: application/json

```
{
  "userID" : "userID"
}
```

Produces

This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.

- `application/json`

Responses

200	sucesseful login loginInfo
400	bad input parameter
401	bad login information

POST /logout

Logout of the platform. (**orgLogout**)

Responses

200	sucesseful logout

POST /event

create new event (**orgNewEvent**)

Consumes

This API call consumes the following media types via the Content-Type request header:

- `application/json`

Request body

newEvent **newEvent** **(required)**
Body Parameter —

Responses

201	success
400	bad input parameter
401	Unauthorized

POST /events/{eventID}/feed

add new feed to an event (**orgNewFeed**)

Path parameters

eventID **(required)**
Path Parameter — Event ID

Consumes

This API call consumes the following media types via the Content-Type request header:

- `application/json`

Request body

newFeed **newFeed** **(required)**
Body Parameter —

Responses
201 success
401 Unauthorized

POST /event/{eventID}/ticketType
create new ticket type (**orgNewType**)

Path parameters
eventID (required)
Path Parameter — Event ID

Consumes
This API call consumes the following media types via the Content-Type request header:
- `application/json`

Request body
newTicketType newTicketType (required)
Body Parameter —

Responses
201 success
400 bad input parameter
401 Unauthorized

PATCH /event/{eventID}/offers/{offerID}
update event offer (currently it only forces revoke) (**orgPatchOffer**)

Path parameters
eventID (required)
Path Parameter — Event ID

offerID (required)
Path Parameter — offerID

Consumes
This API call consumes the following media types via the Content-Type request header:
- `application/json`

Request body
body body (required)
Body Parameter —

Responses
200 success
401 Unauthorized

POST /settings/auth
add authorized email (**orgPutAuth**)

Consumes
This API call consumes the following media types via the Content-Type request header:
- `application/json`

Request body
string array (required)
Body Parameter —

Responses
201 success
400 bad input parameter
401 Unauthorized

PUT /event/{eventID}
update information of an event (**orgPutBaseEvent**)

Path parameters
eventID (required)
Path Parameter — Event ID

Consumes
This API call consumes the following media types via the Content-Type request header:
- `application/json`

Request body
baseEvent [baseEvent] **(required)**
Body Parameter —

Responses
200	success
400	bad input parameter
401	Unauthorized

PUT /settings/mbway
update organization MBWay information (**orgPutMBWay**)

Consumes
This API call consumes the following media types via the Content-Type request header:
- `application/json`

Request body
mbway [mbway] **(required)**
Body Parameter —

Responses
200	success
400	bad input parameter
401	Unauthorized

PUT /settings/profile
update organization profile information (**orgPutProfile**)

Consumes
This API call consumes the following media types via the Content-Type request header:
- `application/json`

Request body
updatableProfile [updatableProfile] **(required)**
Body Parameter —

Responses
200	success
400	bad input parameter
401	Unauthorized

PUT /event/{eventID}/verifiers/{verifierToken}
update verifier information (**orgPutVerifier**)

Path parameters
eventID (required)
Path Parameter — Event ID

verifierToken (required)
Path Parameter — verifier token

Consumes
This API call consumes the following media types via the Content-Type request header:
- `application/json`

Request body
verifierInfo [verifierInfo] **(required)**
Body Parameter —

Responses
200	success
401	Unauthorized

GET /ticketTags

Get tags relative to ticket types (**orgTicketTypeTags**)

Query parameters

lang (required)
Query Parameter — language to retrieve from tags table

Return type

array[tagArray]

Example data

Content-Type: application/json

```
{
  "id" : 0,
  "tag" : "tag"
}
```

Produces

This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.

- `application/json`

Responses

200	returning ticket type tag information
400	bad input parameter
401	Unauthorized

POST /upload

Send images to server (**orgUpload**)

Consumes

This API call consumes the following media types via the Content-Type request header:

- `multipart/form-data`

Form parameters

feed (optional)
Form Parameter — default: null format: binary

cover (optional)
Form Parameter — default: null format: binary

thumbnail (optional)
Form Parameter — default: null format: binary

Return type

imgResponse

Example data

Content-Type: application/json

```
{
  "cover" : "cover",
  "feed" : "feed",
  "thumbnail" : "thumbnail"
}
```

Produces

This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.

- `application/json`

Responses

200	success imgResponse
400	bad input parameter
401	Unauthorized

PUT /event/{eventID}/ticketType/{ticketTypeID}

update ticket type information (**orgUptdateType**)

Path parameters

eventID (required)
Path Parameter — Event ID
ticketTypeID (required)
Path Parameter — Organization ID

Consumes

This API call consumes the following media types via the Content-Type request header:

- `application/json`

Request body

ticketType [ticketType] **(required)**
Body Parameter —

Responses

200	success
400	bad input parameter
401	Unauthorized

DELETE /settings/auth/{email}

delete email from authorized (**settingsAuthEmailDelete**)

Path parameters

email (required)
Path Parameter — Organization ID

Responses

200	success
400	Deletion not avaiable
401	Unauthorized

Models

baseEvent -

name
String

slogan
String

description
String

transports
String

startDate
Date format: date-time

endDate
Date format: date-time

location
location

tags
array[Integer] format: int32

thumbnailImage
String

feedImage
String

coverImage
String

eventSummary -

name
String

thumbnailImage
String

startDate
Date format: date-time

eventID
byte[] format: byte

events -

active
array[eventSummary]

planned
array[eventSummary]

imgResponse -

feed (optional)
String

cover (optional)
String

thumbnail (optional)
String

body -

bulk (optional)
Boolean

eventInfo -

name
String

slogan
String

active
Boolean

description
String

transport
String

startDate
Date format: date-time

endDate
Date format: date-time

location
location

tags
array[Integer] format: int32

ticketTypes
array[ticketTypeSummary]

verifiers
array[verifierSummary]

thumbnailImage
String

feedImage
String

coverImage
String

location -

locationID
Integer format: int32

name
String

address
String

lat
BigDecimal

lng
BigDecimal

type
String

login -

email
String format: email

password
String

rememberMe
Boolean

feed -

eventID
byte[] format: byte

text
String

image
String

feedID
Integer format: int32

publishDate
String

feedArray -

feed
feed

imgUpload -

feed (optional)
File format: binary

cover (optional)
File format: binary

thumbnail (optional)
File format: binary

loginInfo -

userID
byte[] format: byte

mbway -

posID
String

newEvent -

name
String

description
String

startDate
Date format: date-time

endDate
Date format: date-time

newFeed -

text
String

feedUnderscoreimage
String

newOffer -

types
types

email
String format: email

amount (optional)
Integer format: int32

newTicketType -

name
String

description
String

price
Integer format: int32

amount
Integer format: int32

startSaleTime
Date format: date-time

endSaleTime
Date format: date-time

tags
array[Integer] format: int32

secure
Boolean

profile -

name
String

shortName
String

location
String

email
String format: email

contact
String

coverImage
String

feedImage
String

thumbailImage
String

tagArray -

id
Integer format: int32

tag
String

ticketPackage -

ticketPackage_inner -

amount (optional)
Integer format: int32

typeID (optional)
Integer format: int32

name (optional)
String

description (optional)
String

newVerifier -

note
String

offers -

single
array[offers_single]

bulk
array[offers_bulk]

offers_bulk -

email (optional)
String format: email

types (optional)
types

amount (optional)
Integer format: int32

stock (optional)
Integer format: int32

offerID (optional)
byte[] format: byte

packageID (optional)
String

packgeInfo (optional)
ticketPackage

offers_single -

email (optional)
String format: email

types (optional)
types

status (optional)
Integer format: int32

origin (optional)
String

offerID (optional)
byte[] format: byte

packageID (optional)
String

packgeInfo (optional)
ticketPackage

updatableProfile -

location
String

email
String format: email

contact
String

coverImage
String

feedImage
String

thumbailImage
String

ticketType -

name
String

description
String

price
Integer format: int32

amount
Integer format: int32

stock
Integer format: int32

startSaleTime
Date format: date-time

endSaleTime
Date format: date-time

tags
array[Integer] format: int32

secure
Boolean

ticketTypeSummary -

ticketTypeID
byte[] format: byte

name
String

description
String

stock
Integer format: int32

types -

verifierInfo -

note
String

lock
Boolean

tags
array[Integer] format: int32

verifierSummary -

token
byte[] format: byte

note
String

locked
Boolean

Validator API

POST /login

Logins into validator module. (**validatorLogin**)

Consumes

This API call consumes the following media types via the Content-Type request header:

- `application/json`

Request body

login **login** **(required)**
Body Parameter —

Return type

loginInfo

Example data

Content-Type: application/json

```
{
  "verifierID" : "verifierID"
}
```

Produces

This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.

- `application/json`

Responses

200 sucesseful login loginInfo
400 bad input parameter
401 bad login information

POST /logout

Logout. (**validatorLogout**)

Responses

200 sucesseful logout

POST /verify

Sends ticket to validation. (**validatorVerification**)

Consumes

This API call consumes the following media types via the Content-Type request header:

- `application/json`

Request body

ticket **ticket** **(required)**
Body Parameter —

Return type

verifyInfo

Example data

Content-Type: application/json

```
{
  "info" : "info"
}
```

Produces

This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.

- `application/json`

Responses

200 sucesseful attempt at validation verifyInfo
400 bad input parameter

Models

login -

token
String

androidID
String

loginInfo -

verifierID
String

ticket -

ticketKey
String

token (optional)
String

signature
String

verifyInfo -

info
String

Bibliography

[1] Anda aplication. https://play.google.com/store/apps/details?id=pt.opt.anda, September 2018.

[2] Bilheteira online aplication. https://play.google.com/store/apps/details?id=pt.etnaga.controloacessosmobile.bol_controloacessosmobile.staging, September 2018.

[3] Raynair aplication. https://play.google.com/store/apps/details?id=com.ryanair.cheapflights&hl=en, September 2018.

[4] Transpores aereos portugueses aplication. https://play.google.com/store/apps/details?id=com.megasis.android, September 2018.

[5] Whitepaper: A blockchain-based event ticketing protocol. Technical report, Aventus, June 2018.

[6] Techincal documentation of mbway. https://www.mbway.pt/developers/implementacao/, September 2018.

[7] Openapi specification. https://github.com/OAI/OpenAPI-Specification/blob/master/versions/3.0.0.md, September 2018.

[8] Placard aplication. https://play.google.com/store/apps/details?id=pt.scml.placard, September 2018.

[9] Sridhar Balasubramanian, Robert A Peterson, and Sirkka L Jarvenpaa. Exploring the implications of m-commerce for markets and marketing. *Journal of the academy of Marketing Science*, 30(4):348, 2002.

[10] T. Bray. The javascript object notation (json) data interchange format. RFC 7159, RFC Editor, March 2014. http://www.rfc-editor.org/rfc/rfc7159.txt.

[11] Anita Chaudhari, Brinzel Rodrigues, Pratap Sakhare, and Caston Fernandes. Prototype for intelligent ticketing system using nfc. In *Green Computing and Internet of Things (ICGCIoT), 2015 International Conference on*, pages 877–880. IEEE, 2015.

[12] Serge Chaumette, Damien Dubernet, Jonathan Ouoba, Erkki Siira, and Tuomo Tuikka. Architecture and evaluation of a user-centric nfc-enabled ticketing system for small events. In *MobiCASE*, pages 137–151. Springer, 2011.

[13] R. Fielding and J. Reschke. Hypertext transfer protocol (http/1.1): Semantics and content. RFC 7231, RFC Editor, June 2014. http://www.rfc-editor.org/rfc/rfc7231.txt.

[14] Anup K. Ghosh and Tara M. Swaminatha. Software security and privacy risks in mobile e-commerce. *Commun. ACM*, 44(2):51–57, February 2001. ISSN: 0001-0782. doi:10.1145/359205.359227.

[15] William Golden, Eoin Higgins, Martin Hughes, and Susan Flynn. The internet, a creator of electronic markets for airline tickets. In *Proceedings of CoLLECTeR Conference, Galway, Ireland*, 2003.

[16] Gesine Hinterwälder, Christian T. Zenger, Foteini Baldimtsi, Anna Lysyanskaya, Christof Paar, and Wayne P. Burleson. *Efficient E-Cash in Practice: NFC-Based Payments for Public Transportation Systems*, pages 40–59. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. ISBN: 978-3-642-39077-7.

[17] Cosmina Ivan and Roxana Balag. An initial approach for a nfc m-ticketing urban transport system. *Journal of Computer and Communications*, 3(06):42, 2015.

[18] D. M'Raihi, M. Bellare, F. Hoornaert, D. Naccache, and O. Ranen. Hotp: An hmac-based one-time password algorithm. RFC 4226, RFC Editor, December 2005.

[19] D. M'Raihi, S. Machani, M. Pei, and J. Rydell. Totp: Time-based one-time password algorithm. RFC 6238, RFC Editor, May 2011. http://www.rfc-editor.org/rfc/rfc6238.txt.

[20] M. Nystrom and B. Kaliski. Pkcs #10: Certification request syntax specification version 1.7. RFC 2986, RFC Editor, November 2000.

[21] E. O'Tuathail and M. Rose. Using the simple object access protocol (soap) in blocks extensible exchange protocol (beep). RFC 4227, RFC Editor, January 2006.

[22] M. Magdalena Payeras-Capellà, Macià Mut-Puigserver, Josep-Lluís Ferrer-Gomila, Jordi Castellà-Roca, and Arnau Vives-Guasch. *Electronic Ticketing: Requirements and Proposals Related to Transport*, pages 285–301. Springer International Publishing, Cham, 2015. ISBN: 978-3-319-09885-2.

[23] Keng Siau and Zixing Shen. Building customer trust in mobile commerce. *Commun. ACM*, 46(4):91–94, April 2003. ISSN: 0001-0782. doi:10.1145/641205.641211.